September 24, 2024

New Zealand Informatics Competition 2024

Round Three

Editorial by Anatol C, Zalan V, Jonathon S

Contents

Introduction 2
Resources 2
Chess 3
Duckstop 4
Rome 5
Word Ladder 7
A: Drive 9
Big O Complexity 12
Additional Example Solutions 13

Introduction

The solutions to this round of NZIC problems are discussed in detail below. In a few questions we may refer to the Big O complexity of a solution, e.g. $O(N)$. There is an explanation of Big O complexity at the end of this document.

Resources

Ever wondered what the error messages mean?

https://www.nzoi.org.nz/nzic/resources/understanding-judge-feedback.pdf

Read about how the server marking works:

https://www.nzoi.org.nz/nzic/resources/how-judging-works-python3.pdf

Ever wondered why your submission scored zero?

https://www.nzoi.org.nz/nzic/resources/why-did-i-score-zero.pdf

See our list of other useful resources here:

https://www.nzoi.org.nz/nzic/resources

Chess

Problem authored by Iván G
https://train.nzoi.org.nz/problems/1431

Subtask 3

We can create a 500×500 2D array storing the colour of each cell and use it to answer every query.

Full Solution

If $a + b$ is even, then it is a black square, otherwise it is a white square.

Proof: We begin at square $(0, 0)$, which is black. After moving a squares to the right, the colour is flipped if and only if a is odd. Similarly, after moving b squares upwards, the colour is flipped if and only if b is odd. Let p and q denote the parities of a and b respectively (with 0 being even and 1 being odd). The final colour is black if and only if p XOR $q = 0$, which is equivalent to checking if $a + b$ is even.

Python Solution

```
n = int(input())
for _ in range(n):
    a, b = list(map(int, input().split()))
    print("WHITE" if (a + b) % 2 == 1 else "BLACK")
```

Duckstop

Problem authored by Zalan V
https://train.nzoi.org.nz/problems/1446

Subtask 1

There are only two cases:

1. We can afford at least one duck on day 1, and the price is at most A dollars, and will be able to sell them for a higher price on day 2. In this case, we should BUY as many ducks as we can on day 1 and SELL them all on day 2.
2. In every other situation, we must WAIT on both days and make a profit of 0 dollars.

Python Solution

```
D, C, A = map(int, input().split())
prices = list(map(int, input().split()))
if prices[0] <= min(A, C) and prices[0] < prices[1]:
    buy_amount = C // prices[0]
    print("BUY", buy_amount)
    print("SELL", buy_amount)
    print(buy_amount * (prices[1] - prices[0]))
else:
    print("WAIT")
    print("WAIT")
    print(0)
```

Subtask 2

For this subtask, we can simulate the entire process. There are two states that we can be in:

1. We currently have no ducks. In this state, we check if the price of ducks is at most A and if we can afford at least one duck and will be able to sell them for a higher price later. If all three of these conditions are met then we BUY as many ducks as we can and switch to state 2. Otherwise, WAIT.
2. We currently have at least one duck. In this state, check if the current price is higher than the price of the ducks that we most recently bought. If it is, then SELL all our ducks and switch to state 1. Otherwise, HOLD.

The time complexity of this solution is $O(D^2)$ since to find the maximum price after a day takes $O(D)$ time.

Python Solution

The code for this solution can be found at the end of this document.

Full Solution

For the full solution we need to efficiently check whether there exists a later day with a higher price.

We can do this by creating a suffix maximum array before starting the simulation. Let s denote this array and p denote the prices array. If we iterate through the days in reverse order, then for each day i, $s[i] = \max(p[i], s[i+1])$. This means that for each i, $s[i]$ is equal to $\max(p[i], p[i+1], ..., p[D])$. During the simulation, we can just check if $s[i] > p[i]$ to know whether or not we can sell for a higher price later. The time complexity of this solution is then $O(D)$.

Python Solution

The code for this solution can be found at the end of this document.

Rome

Problem authored by Iván G
https://train.nzoi.org.nz/problems/1389

Subtask 1

For on each day, for every tower, we can iterate to the left and right, until we either find a tower of equal height at which point we know it is safe, or a tower that is taller which means that we can stop as no tower will be visible after.

Time complexity: $O(N^3)$

Python Solution

The code for this solution can be found at the end of this document .

Subtask 2

We consider that if a tower becomes safe on some particular day, then it will also be safe on every day afterwards. If a tower of equal height is visible to the left then a tower will become safe on the day that it is built, otherwise if a tower will be visible on the right then it will become safe on the day that the tower to the right is built. Otherwise the tower will never be safe. Thus, we can find when a tower will become safe (if ever) in $O(N)$.

We maintain an array of the number of unsafe watchtowers on each day, which we intialise to be $1, 2, 3, \ldots$. Then, for each watchtower we subtract -1 from every day from when it becomes safe. We can do this on $O(N)$ for each tower.

The total time complexity of this solution is then $O(N^2)$.

Python Solution

The code for this solution can be found at the end of this document .

Subtask 3

In this subtask, h_i is at most 10, so we can iterate through the towers for each possible height (1 to 10) and maintain whether there is a visible tower to the left. For each tower that matches the current height we can store the first day that it becomes safe. This is similar to Subtask 2, however we must add 1 to only $s[i]$ only. After all heights have been processed, we can perform a prefix sum on s, i.e. $s[i] = s[i] + s[i-1]$ for each i from 0 to $N-1$.

Time complexity: $O(\max(h[i]) \times N)$

Python Solution

The code for this solution can be found at the end of this document .

Subtask 4 & 5 (Full Solution)

For the full solution we maintain an array of all towers that are potentially visible on each day. We consider that if a tower a is built that is taller than some tower b in the array, then tower b will never be visible and so we can remove it. We realise that if we always remove no longer visible towers from the array and add the current tower to the end then we will always remove some suffix of the array. Furthermore, this array will be decreasing. This data structure is known as a monotonic stack, and has an amortised time complexity of $O(1)$ (over all N additions to the array that total runtime will be $O(N)$).

Then, we can efficiently check if a newly built tower is visible from some other tower. We also need to handle the case of whether a tower is already safe or becomes safe after it is visible from a newly built tower.

Time complexity: $O(N)$

Alternatively, you can use any data structure that supports efficient range maximum queries (such as sparse tables or segment trees) to solve it in $O(N \log N)$. In C++, this problem can also be solved using ordered sets in $O(N \log N)$.

Python Solution

The code for this solution can be found at the end of this document .

Word Ladder

Problem authored by Anatol C
https://train.nzoi.org.nz/problems/1370

Subtask 1

In this subtask, there are no valid words apart from the start or end word. Let x be the number of letters we need to change to get from turn the start word into the end word. If $x = 1$ then, that means we can always go from the start word to the end word in a single step. If $x = 2$, we would need to add one extra word to be able to step from the start word, to the new word, to the end word.

More generally, for any value of x, we can win the game in x steps by adding $x - 1$ words. This is always the optimal way of winning in this subtask. To be able to add $x - 1$ extra words, we need K to be at least $x - 1$. This leads to the following solution:

- If $K \geq x - 1$, it is possible to win in x steps (but no less).
- Otherwise, Jon cannot win.

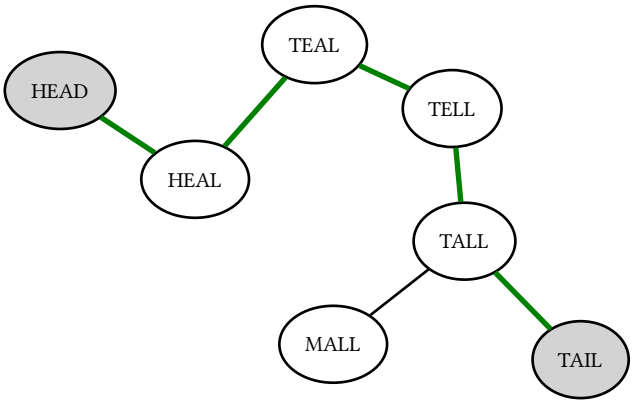
Python Solution

```
N, M, K = map(int, input().split())
S = input()
E = input()
x = 0
for i in range(N):
  if S[i] != E[i]:
    x += 1
if K >= x-1:
  print(x)
else:
  print("IMPOSSIBLE")
```

Subtask 2

In this subtask, we cannot add any extra words, so we just need to find the fastest way to win with the current words.

We can transform this into a graph problem. Construct a graph where every valid word is a node and add a connection between any two nodes if we can step from one word to the other (that is, they differ by at most one letter). Then, if there is a path from the start word to the end word, the answer is the length of the shortest such path, otherwise it is impossible for Jon to win. The graph for sample 1 is given as an example below.



We can compute the shortest path using a breadth-first search.

Python Solution
The code for this solution can be found at the end of this document .

Subtask 3

In this subtask, we keep using the graph described above. If $K = 0$, then the solution is as above, the shortest distance between the start and end nodes in the graph.

If $K = 1$, we can add a single new node, we want this node to be able to connect to (at least) two other nodes. Otherwise, this node can't possibly form a shorter path. Call the two nodes our two nodes can connect with a and b. Our new node must differ by at most one letter from a and b. This means a and b must differ by exactly one or two letters. In the case that they differ by exactly one letter, adding the new node is pointless since it cannot offer a shorter path between these two nodes than what already exists. So, we only care about the case where a and b differ by exactly two letters.

We want to consider all possible new nodes we can add that satisfy the criteria outlined above. To do so, we can iterate over all pairs of nodes a and b and check that they differ by exactly most two letters in $O(M^2N)$. Then, for each pair, we want to calculate the shortest path that would go through our new node, and then our solution is the minimum value over all these pairs. We can efficiently compute the length of the new shortest path. If we precompute the distance from start to all nodes (as $\text{dist}_S[i]$), and from end to all nodes (as $\text{dist}_E[i]$), then the length of our new paths is just:

$$\min(2 + \text{dist}_S[a] + \text{dist}_E[b], 2 + \text{dist}_S[b] + \text{dist}_E[a])$$

Note that some of the distance values may be infinite, denoting that no corresponding path exists.

Python Solution
The code for this solution can be found at the end of this document .

Subtask 4

This subtask exists to allow inefficient versions of the full solution to get partial marks.

Subtask 5

For a full solution, we need to rethink how we construct our graph to represent the problem. We can construct a graph were every node is a tuple (w, x) where w is a valid word and x is an integer between 0 and K. Such a node represents getting to word w by adding exactly x extra words. More specifically, we want the distance from $(S, 0)$ to (w, x) to be the minimum amount of steps needed to get from the starting word to word w by adding exactly x extra words.

It is simpler to construct the graph implicitly. Let $\text{diff}(a, b)$ represent the number of letters in word a we need to change to get word b. From a node (w, x), and for any word w', there is an edge from (w, x) to $(w', x + \text{diff}(w, w') - 1)$ with length $\text{diff}(w, w')$, given that $x + \text{diff}(w, w') - 1 \leq K$.

Finally, we can use Dijkstra's algorithm to find the shortest path from $(S, 0)$ to any (E, x) node or find that such that a path does not exists. Note that we can (and should) speed up our solution by a factor of N by precomputing all the $\text{diff}(a, b)$ values.

Python Solution
The code for this solution can be found at the end of this document .

C++ Solution
The code for this solution can be found at the end of this document .

A: Drive

Problem authored by Zalan V
https://train.nzoi.org.nz/problems/1448

Subtask 1

In this subtask there are only C and O type events. C type events will create files contiguously starting from the first disk, so the O type events will have no impact.

Python Solution

```
N, M = map(int, input().split())
first_free = 1
for i in range(N):
    inp = input().split()
    if inp[0] == 'C':
        print(first_free)
        first_free += (int(inp[1]) + 1474560 - 1) // 1474560
    else:
        print(0)
```

Subtask 2

In this subtask there are at most 100 disks, and $N \leq 10,000$ so it will be sufficient to process all N queries in $O(M)$. We can maintain an array of the M disks, where each empty disk will be labelled with 0 if it is empty, otherwise with the id of the file on the disk.

To create a file we can find the first contiguous range of empty disks that is large enough by scanning through all disks, and set the value of each of these disks to the file id. To delete a file we can scan through and mark each of the disks occupied by the file as free by setting it to 0. To modify a file we first find the file and the free space directly after it, and modify it in place if the new file size will fit, otherwise we delete and create the file. For optimisation queries we can loop through the disks and move each occupied disk to the lowest free disk and keep track of how many disks we have moved. Each of these operations can be implemented in $O(M)$. This leads to a total complexity of $O(NM)$ for this solution.

Python Solution

The code for this solution can be found at the end of this document .

Subtask 3

We can represent files as a sorted array of (position, length, file_id). Then, to create a file we can loop through the array and find a gap large enough for the new file, and insert the new file at the required position. We can delete a file by removing it from its position in the array. To modify a file we can simply change its size if it will fit, otherwise delete then create it again. For optimisation queries we can loop through and update the position of each file as needed. Let F be the maximum number of existing files at any time. Then, we can perform each of these events in $O(F)$ and so this solution will have a complexity of $O(NF)$.

We can observe that F is at most 100 in Subtask 2, since each file must occupy at least one disk, so this solution will also pass Subtask 2. In Subtask 1 we know that optimisation queries have no impact. We can generalise this to the following: an optimisation query will only have an impact if a modify or delete query has occurred. Furthermore, if the file layout is optimised, then we know that the file will go directly after the last file. By adding these optimisations, this solution will pass Subtask 1 and will score 55 points.

Python Solution

The code for this solution can be found at the end of this document .

Subtask 4

For this subtask we can store a segment tree over an array of size M in which we keep track of the number of free disks at the starting position of every free segment. Then, we need to be able to query the rightmost non-zero value, and the maximum value in a range to process all types of events.

We can find the position to create a file in $O(\log N)$ using a segment tree walk. At each node we walk to the left child if the maximum value is large enough, otherwise we walk to the right child. Alternatively, we could also use a binary search to find the first prefix that includes a free space of suitable size in $O(\log^2 N)$.

Deleting a file involves potentially merging free ranges directly before and after the file into one free range. We can easily find the length (if any) of the free range after the file, as it will be in the position directly after then end of the file if it exists (we also need to consider the case where the file ends on disk M). To find the free range before the file we query the rightmost non-zero value before the file, and check if it reaches the file.

Modification requires a query to find the available space directly after the file. If there is sufficient space then we can move the free range as needed. Otherwise we delete + create the file.

The complexity of this solution is $O(N \log N)$.

C++ Solution

The code for this solution can be found at the end of this document .

Subtask 5

This subtask does not differ significantly from Subtask 4, but is slightly more difficult to implement since we must also consider optimistaion queries. Since there are at most 10 O events we do not need to process optimisation queries particularly efficiently: it is sufficient to process each O type event in $O(N)$ or $O(N \log N)$.

Also including the optimisation to skip O type events which have no impact will mean that this solution will pass subtasks 1 through 5 and will score 97 points. This was the original intended full solution, but in the end an extra subtask was added as a bonus challenge.

C++ Solution

The code for this solution can be found at the end of this document .

Full Solution

This subtask was intended as an extra challenge for those who solved everything else in the contest. Congratulations to the single contestant who solved the full problem in contest.

The idea for the full solution builds upon the solution for Subtask 3. Namely, we store files in some type of Self Balancing Binary Search Tree (SBBST) which could include treaps, red-black trees, AVL trees, etc. In this editorial we will cover a solution using a modified AVL tree. Information on AVL trees can be found here: https://en.wikipedia.org/wiki/AVL_tree

To facilitate optimisation events we will use lazy propagation. We will refer to a compressed subtree as a subtree which has been optimised, so the files have all been tightly packed (i.e. all files have been shifted left enough that there is no longer any empty disks). Since we are using lazy propagation, we will need to take extra care to propagate the compressed status, especially during tree balancing rotations.

For each node in the tree we need to store:

- Parent
- Left child
- Right child
- Subtree height
- Position
- Size
- The leftmost occupied disk in subtree
- The rightmost occupied disk in subtree
- The total number of occupied disks in subtree
- The longest contiguous free range of disks in the subtree
- A flag to indicate whether the current subtree has been compressed

The idea is that a node will only "own" the position and size fields, and all other fields are simply precalculated values for the subtree.

To create a file we recurse from the root, keeping track of the free space before and after the subtree. At each node we first check if we can create the file within the left subtree. There are three cases for when the file will fit in the left subtree:

1. The file will fit in the free space before the left subtree
2. The file will fit within the left subtree
3. The file will fit within the space between the left subtree and the current node

We can check these efficiently using the values in each node. Otherwise, we recurse to the right subtree. Once we have created the file, we move back up the tree and rebalance and update nodes as required. We then store a pointer to the newly create node representing the file, which will be used for modify and delete events. The complexity of modifying a file is thus $O(\log N)$.

To delete a file, we first need to find the position of the file. Due to optimisation events, some ancestors of the node are potentially compressed (which would change the position) so we recurse upwards to the root and decompress the ancestors in order down to the current node. Then, we can delete the node as we would in a standard AVL tree, although we need to take care in the implementation to move nodes around in the tree rather than create copies, as we have pointers to nodes that need to stay valid. The complexity of deleting a file is thus $O(\log N)$.

To modify a file we also need to first decompress all ancestors. Then, we can recurse from the root and keep track of the rightmost boundary of the free space directly after the current subtree. We can use this to determine whether we can extend the file in place. In the case that we can modify the file in place, we simply update the node and update (no rebalance required since no nodes are created or deleted) nodes back up to the root. In the case that the file no longer fits we simply delete and create the file. The complexity of modifying a file is thus $O(\log N)$.

Handling optimisation events thus becomes as simple as checking the difference between the rightmost disk of the root and it's total size, and compressing the root. This is then $O(1)$.

Overall the complexity of this solution is $O(N \log N)$.

C++ Solution

The code for this solution can be found at the end of this document .

Big O Complexity

Computer scientists like to compare programs using something called Big O notation. This works by choosing a parameter, usually one of the inputs, and seeing what happens as this parameter increases in value. For example, let's say we have a list N items long. We often call the measured parameter N. For example, a list of length N.

In contests, problems are often designed with time or memory constraints to make you think of a more efficient algorithm. You can estimate this based on the problem's constraints. It's often reasonable to assume a computer can perform around 100 million (100,000,000) operations per second. For example, if the problem specifies a time limit of 1 second and an input of N as large as 100,000, then you know that an $O(N^2)$ algorithm might be too slow for large N since $100{,}000^2 = 10{,}000{,}000{,}000$, or 10 billion operations.

Time Complexity

The time taken by a program can be estimated by the number of processor operations. For example, an addition $a + b$ or a comparison $a < b$ is one operation.

$O(1)$ time means that the number of operations a computer performs does not increase as N increases (i.e. does not depend on N). For example, say you have a program containing a list of N items and want to access the item at the i-th index. Usually, the computer will simply access the corresponding location in memory. There might be a few calculations to work out which location in memory the entry i corresponds to, but these will take the same amount of computation regardless of N. Note that time complexity does not account for constant factors. For example, if we doubled the number of calculations used to get each item in the list, the time complexity is still $O(1)$ because it is the same for all list lengths. You can't get a better algorithmic complexity than constant time.

$O(\log N)$ time suggests the program takes a constant number of extra operations every time N doubles in size. For example, finding a number in a sorted list using binary search might take 3 operations when $N = 8$, but it will only take one extra operation if we double N to 16. As far as efficiency goes, this is pretty good, since N generally has to get very, very large before a computer starts to struggle.

$O(N)$ time means you have an algorithm where the number of operations is directly proportional to N. For example, a maximum finding algorithm `max()` will need to compare against every item in a list of length N to confirm you have indeed found the maximum. Usually, if you have one loop that iterates N times your algorithm is $O(N)$.

$O(N^2)$ time means the number of operations is proportional to N^2 . For example, suppose you had an algorithm which compared every item in a list against every other item to find similar items. For a list of N items, each item has to check against the remaining $N - 1$ items. In total, $N(N - 1)$ checks are done. This expands to $N^2 - N$. For Big O, we always take the most significant term as the dominating factor, which gives $O(N^2)$. This is generally not great for large values of N, which can take a very long time to compute. As a general rule of thumb in contests, $O(N^2)$ algorithms are only useful for input sizes of $N \lesssim 10{,}000$. Usually, if you have a nested loop in your program (loop inside a loop) then your solution is $O(N^2)$ if both these loops run about N times.

Additional Example Solutions

Duckstop Subtask 2 Python Solution

```
D, C, A = map(int, input().split())
stock = 0
last_bought_at = 0
cash = C
prices = list(map(int, input().split()))
for i in range(D):
    if stock:
        if prices[i] > last_bought_at:
            print("SELL", stock)
            cash += stock * prices[i]
            stock = 0
        else:
            print("HOLD")
    elif prices[i] <= min(cash, A) and prices[i] < max(prices[i:]):
        stock = cash // prices[i]
        cash %= prices[i]
        last_bought_at = prices[i]
        print("BUY", stock)
    else:
        print("WAIT")
print(cash - C)
```

Duckstop Full Python Solution

```
D, C, A = map(int, input().split())
stock = 0
last_bought_at = 0
cash = C
prices = list(map(int, input().split()))

max_future = list(prices)
for i in range(D - 2, -1, -1):
    max_future[i] = max(max_future[i], max_future[i + 1])

for i in range(D):
    if stock:
        if prices[i] > last_bought_at:
            print("SELL", stock)
            cash += stock * prices[i]
            stock = 0
        else:
            print("HOLD")
    elif prices[i] <= min(cash, A) and prices[i] < max(max_future[i]):
        stock = cash // prices[i]
        cash %= prices[i]
        last_bought_at = prices[i]
        print("BUY", stock)
    else:
        print("WAIT")
print(cash - C)
```

Rome Subtask 1 Python Solution

```
n = int(input())
h = list(map(int, input().split()))
```

```
for i in range(n):
    num_safe = 0
    for j in range(i+1):
        is_safe = 0
        # Check left
        cur = j-1
        while cur >= 0 and h[cur] < h[j]:
            cur -= 1
        if cur >= 0 and h[cur] == h[j]:
            is_safe = 1
        # Check right
        cur = j+1
        while cur <= i and h[cur] < h[j]:
            cur += 1
        if cur <= i and h[cur] == h[j]:
            is_safe = 1
        num_safe += is_safe
    print(i + 1 - num_safe)
```

Rome Subtask 2 Python Solution

```
n = int(input())
h = list(map(int, input().split()))
s = [0] * n

def check_left(x):
    cur = x-1
    while cur >= 0 and h[cur] < h[x]:
        cur -= 1
    if cur >= 0 and h[cur] == h[x]:
        return x
    return n+1

def check_right(x):
    cur = x+1
    while cur < n and h[cur] < h[x]:
        cur += 1
    if cur < n and h[cur] == h[x]:
        return cur
    return n+1

for i in range(n):
    first_safe = min(check_left(i), check_right(i))
    for j in range(first_safe, n):
        s[j] += 1

for i in range(n):
    print(i + 1 - s[i])
```

Rome Subtask 3 Python Solution

```
n = int(input())
h = list(map(int, input().split()))
s = [0] * n
first = [n+1] * n

for height in range(1, 11):
```

```
    cur = -1
    for i in range(n):
        if h[i] == height:
            if cur != -1:
                first[cur] = min(first[cur], i)
                first[i] = i
            cur = i
        elif h[i] > height:
            cur = -1

for i in range(n):
    if first[i] < n:
        s[first[i]] += 1

for i in range(n):
    if i > 0:
        s[i] += s[i-1]
    print(i + 1 - s[i])
```

Rome Full Python Solution

```
n = int(input())
h = list(map(int, input().split()))

safe = [0] * n
stack = []
cnt = 0
for i in range(n):
    while stack and h[i] > h[stack[-1]]:
        stack.pop()
    if stack and h[i] == h[stack[-1]]:
        safe[i] = 1
        cnt += 1
        if not safe[stack[-1]]:
            safe[stack[-1]] = 1
            cnt += 1
        stack.pop()
    stack.append(i)
    print(i + 1 - cnt)
```

Word Ladder Subtask 2 Python Solution

```
from collections import deque
N, M, K = map(int, input().split())
S = input()
E = input()

valid_words = [input() for _ in range(M)]

edges = [[] for _ in range(M)]
for i in range(M):
    for j in range(i+1, M):
        diff = 0
        for k in range(N):
            if valid_words[i][k] != valid_words[j][k]:
                diff += 1
        if diff == 1:
            edges[i].append(j)
```

```
            edges[j].append(i)

q = deque()
start, end = valid_words.index(S), valid_words.index(E)
q.append((start, 0))

visited = [False] * M
while len(q):
    node, dist = q.popleft()
    if visited[node]: continue
    visited[node] = True
    if node == end:
        print(dist)
        exit(0)
    for c in edges[node]:
        q.append((c, dist + 1))

print("IMPOSSIBLE")
```

Word Ladder Subtask 3 Python Solution

```
from collections import deque
from math import isinf

N, M, K = map(int, input().split())
S = input()
E = input()

valid_words = [input() for _ in range(M)]

edges = [[] for _ in range(M)]
for i in range(M):
    for j in range(i+1, M):
        diff = 0
        for k in range(N):
            if valid_words[i][k] != valid_words[j][k]:
                diff += 1
        if diff == 1:
            edges[i].append(j)
            edges[j].append(i)

start, end = valid_words.index(S), valid_words.index(E)

q = deque()
q.append((start, 0))
dist_start = [float("inf")] * M
while len(q):
    node, dist = q.popleft()
    if not isinf(dist_start[node]): continue
    dist_start[node] = dist
    for c in edges[node]:
        q.append((c, dist + 1))

q = deque()
q.append((end, 0))
dist_end = [float("inf")] * M
while len(q):
```

```
    node, dist = q.popleft()
    if not isinf(dist_end[node]): continue
    dist_end[node] = dist
    for c in edges[node]:
        q.append((c, dist + 1))

res = dist_start[end]

if K != 0:
    for a in range(M):
        for b in range(M):
            diff = 0
            for k in range(N):
                if valid_words[a][k] != valid_words[b][k]:
                    diff += 1
            if diff == 2:
                res = min(res, min(2 + dist_start[a] + dist_end[b], 2 + dist_start[b]
+ dist_end[a]))

if isinf(res): print("IMPOSSIBLE")
else: print(res)
```

Word Ladder Python Full Solution

```
import heapq

N, M, K = map(int, input().split())
S = input()
E = input()

words = [input() for _ in range(M)]
diff = [[0] * M for _ in range(M)]
v = [[False] * 105 for _ in range(105)]

for i in range(M):
    for j in range(M):
        for l in range(N):
            if words[i][l] != words[j][l]:
                diff[i][j] += 1

si = words.index(S)
ei = words.index(E)

pq = []
heapq.heappush(pq, (0, (si, 0)))

while len(pq):
    d, (word, x) = heapq.heappop(pq)
    if v[word][x]: continue
    v[word][x] = True

    if word == ei:
        print(d)
        exit(0)

    for i in range(M):
        if i == word: continue
```

```
        if x + diff[word][i] - 1 <= K:
            heapq.heappush(pq, (d + diff[word][i], (i, x + diff[word][i] - 1)))

print("IMPOSSIBLE")
```

Word Ladder C++ Full Solution

```
#include <bits/stdc++.h>

using namespace std;

int n, m, k;
string s, e, words[100];
int si, ei;

int v[105][105];
int dist[100][100];
int origins[105][105];

int main() {
    cin >> n >> m >> k >> s >> e;

    for (int i = 0; i < m; i++) {
        cin >> words[i];
        if (words[i] == s) si = i;
        else if (words[i] == e) ei = i;
    }

    for (int i = 0; i < m; i++) {
        for (int j = 0; j < m; j++) {
            for (int k = 0; k < n; k++) {
                if (words[i][k] != words[j][k]) dist[i][j]++;
            }
        }
    }

    priority_queue<pair<int, pair<pair<int, int>, int>>> pq;
    pq.push({0, {{si, k}, -1}});

    while (pq.size()) {
        int d = -pq.top().first;
        int word = pq.top().second.first.first;
        int wordsLeft = pq.top().second.first.second;
        int origin = pq.top().second.second;
        pq.pop();

        if (v[word][wordsLeft]) continue;
        v[word][wordsLeft] = true;

        origins[word][wordsLeft] = origin;

        if (word == ei) {
            cout << d << endl;
            return 0;
        }

        for (int i = 0; i < m; i++) {
```

```
            if (i == word) continue;

            int needed = dist[word][i] - 1;
            if (wordsLeft < needed) continue;

            pq.push({-(d + needed + 1), {{i, wordsLeft - needed}, word}});
        }
    }

    cout << "IMPOSSIBLE" << endl;
}
```

A: Drive Subtask 2 Python Solution

```
N, M = map(int, input().split())
disks = [0] * M

def get_size(size_bytes):
    return (size_bytes + 1474560 - 1) // 1474560

def create_file(size, file_id):
    # Find first free range of at least size
    first_free = 0
    for i in range(M):
        if disks[i]:
            first_free = i + 1
        elif i - first_free + 1 == size:
            break
    # Mark disks as used
    for i in range(first_free, first_free + size):
        disks[i] = file_id
    return first_free + 1

def delete_file(file_id):
    idx = disks.index(file_id)
    for i in range(idx, M):
        if disks[i] != file_id:
            break
        disks[i] = 0
    return idx + 1

next_file_id = 1
for _ in range(N):
    query = input().split()
    if query[0] == 'C':
        print(create_file(get_size(int(query[1])), next_file_id))
        next_file_id += 1
    elif query[0] == 'D':
        print(delete_file(int(query[1])))
    elif query[0] == 'M':
        file_id = int(query[1])
        new_size = get_size(int(query[2]))
        idx = disks.index(file_id)
        cur_size = disks.count(file_id)
        # Check if the file has shrunk or stayed the same size
        if new_size <= cur_size:
            for i in range(idx + new_size, idx + cur_size):
```

```
                disks[i] = 0
            print(idx + 1)
        else:
            # Find the number of contiguous free disks directly after the file
            free_disks = 0
            for i in range(idx + cur_size, M):
                if disks[i]:
                    break
                free_disks += 1
            # Extend the file if it fits
            if new_size <= cur_size + free_disks:
                for i in range(idx + cur_size, idx + new_size):
                    disks[i] = file_id
                print(idx + 1)
            # Recreate the file
            else:
                delete_file(file_id)
                print(create_file(new_size, file_id))
    else:
        # Find the last occupied disk, which we will use to measure the optimisation
        for last_occupied in range(M - 1, -1, -1):
            if disks[last_occupied]:
                break
        else:
            # No used disks
            print(0)
            continue
        # Move every occupied disk to the next lowest disk
        first_free = 0
        for i in range(M):
            if disks[i]:
                if first_free < i:
                    disks[first_free] = disks[i]
                    disks[i] = 0
                first_free += 1
        # Find the change in the last disk to measure the optimisation
        for new_last_occupied in range(M - 1, -1, -1):
            if disks[new_last_occupied]:
                print(last_occupied - new_last_occupied)
                break
```

A: Drive Subtask 3 Python Solution

```
N, M = map(int, input().split())
files = []
next_file_ID = 1
optimised = True

def get_disk_size(a):
    return (a + 1474560 - 1) // 1474560

def file_create(size, file_ID):
    if len(files) == 0:
        files.append([1, size, file_ID])
        return 1

    if optimised:
```

```
            files.append([files[-1][0] + files[-1][1], size, file_ID])
            return files[-1][0]

        last = 1
        for i in range(len(files)):
            if files[i][0] - last >= size:
                files.insert(i, [last, size, file_ID])
                return last
            last = files[i][0] + files[i][1]

        loc = files[-1][0] + files[-1][1]
        files.append([loc, size, file_ID])
        return loc

    for _ in range(N):
        query = input().split()
        if query[0] == 'C':
            print(file_create(get_disk_size(int(query[1])), next_file_ID))
            next_file_ID += 1
        elif query[0] == 'D':
            optimised = False
            file_ID = int(query[1])
            for i in range(len(files)):
                if files[i][2] == file_ID:
                    print(files.pop(i)[0])
                    break
        elif query[0] == 'M':
            optimised = False
            file_ID = int(query[1])
            fsz = get_disk_size(int(query[2]))
            if files[-1][2] == file_ID:
                if M - files[-1][0] >= fsz:
                    files[-1][1] = fsz
                    print(files[-1][0])
                else:
                    files.pop()
                    print(file_create(fsz, file_ID))
                continue
            for i in range(len(files) - 1):
                if files[i][2] == file_ID:
                    if files[i + 1][0] - files[i][0] >= fsz:
                        files[i][1] = fsz
                        print(files[i][0])
                    else:
                        files.pop(i)
                        print(file_create(fsz, file_ID))
                    break
        else:
            if len(files) == 0 or optimised:
                print(0)
                continue
            optimised = True
            curr = files[-1][0]
            counter = 1
            for i in range(len(files)):
                files[i][0] = counter
```

```
            counter += files[i][1]
        print(curr - files[-1][0])
```

A: Drive Subtask 4 C++ Solution

```
#include <bits/stdc++.h>
using namespace std;
typedef long long ll;
typedef pair<int, int> ii;


int N, M;
unordered_map<int, ii> files;


struct node {
    ii vr; // Rightmost value {value, position}
    ii vm; // Maximum value {value, position}
};

ii combineRightmost(const ii& left, const ii& right) {
    return right.first ? right : left;
}

ii combineMaximum(const ii& left, const ii& right) {
    return left.first >= right.first ? left : right;
}

node combine(const node& left, const node& right) {
    return {combineRightmost(left.vr, right.vr), combineMaximum(left.vm, right.vm)};
}

struct segtree {
    vector<node> nodes;

    segtree(int s): nodes(s * 4) {}

    void update(int n, int tl, int tr, int pos, int val) {
        if (tl == tr) {
            nodes[n] = {{val, tl}, {val, tl}};
            return;
        }
        int tm = (tl + tr) / 2;
        if (pos <= tm) update(n * 2, tl, tm, pos, val);
        else update(n * 2 + 1, tm + 1, tr, pos, val);
        nodes[n] = combine(nodes[n * 2], nodes[n * 2 + 1]);
    }

    ii queryRightmost(int n, int tl, int tr, int l, int r) {
        if (l <= tl && tr <= r) return nodes[n].vr;
        int tm = (tl + tr) / 2;
        ii rl{}, rr{};
        if (l <= tm) rl = queryRightmost(n * 2, tl, tm, l, r);
        if (tm < r) rr = queryRightmost(n * 2 + 1, tm + 1, tr, l, r);
        return combineRightmost(rl, rr);
    }
```

```
    ii getFilePos(int n, int tl, int tr, int len) {
        if (tl == tr) return nodes[n].vm;
        int tm = (tl + tr) / 2;
           return nodes[n * 2].vm.first >= len ? getFilePos(n * 2, tl, tm, len) :
getFilePos(n * 2 + 1, tm + 1, tr, len);
    }
};

int fileCreate(segtree& tree, int fileID, int len) {
    ii frr = tree.getFilePos(1, 1, M, len);
    int pos = frr.second;
    files[fileID] = {pos, len};
    tree.update(1, 1, M, pos, 0);
    int rem = frr.first - len;
    if (rem) tree.update(1, 1, M, pos + len, rem);
    return pos;
}

int fileDelete(segtree& tree, int fileID) {
    int pos = files[fileID].first;
    int len = files[fileID].second;
    files.erase(fileID);
    ii vprev = tree.queryRightmost(1, 1, M, 1, pos);
    ii vnext{};
    if (pos + len <= M) vnext = tree.queryRightmost(1, 1, M, pos + len, pos + len);

    if (vnext.first) {
        tree.update(1, 1, M, pos + len, 0);
        len += vnext.first;
    }

   if (vprev.second + vprev.first == pos) tree.update(1, 1, M, vprev.second, vprev.first
+ len);
    else tree.update(1, 1, M, pos, len);
    return pos;
}

int getDiskSize(ll a) { return (a + 1474560 - 1) / 1474560; }

int main() {
    // Fast IO
    cin.tie(0);
    ios_base::sync_with_stdio(false);

    cin >> N >> M;

    int nextFileID = 1;
    bool optimised = true;

    segtree tree(M);
    // Initially all disks are free
    tree.update(1, 1, M, 1, M);

    char a;
    int b;
    ll c;
```

```
    while (N--) {
        cin >> a;
        if (a == 'C') {
            cin >> c;
            cout << fileCreate(tree, nextFileID++, getDiskSize(c)) << "\n";
        }
        else if (a == 'D') {
            cin >> b;
            cout << fileDelete(tree, b) << "\n";
        }
        else if (a == 'M') {
            cin >> b >> c;
            int newLen = getDiskSize(c);
            int pos = files[b].first;
            int len = files[b].second;

            // File size has not changed
            if (len == newLen) cout << pos << "\n";
            else {
                ii vnext{};
                int end = pos + len;
                if (end <= M) vnext = tree.queryRightmost(1, 1, M, end, end);
                // File has shrunk
                if (newLen < len) {
                    if (vnext.first) tree.update(1, 1, M, end, 0);
                    tree.update(1, 1, M, pos + newLen, len + vnext.first - newLen);
                    cout << pos << "\n";
                }
                else {
                    // File can be extended
                    if (newLen - len <= vnext.first) {
                        tree.update(1, 1, M, end, 0);
                        int rem = len + vnext.first - newLen;
                        if (rem) tree.update(1, 1, M, pos + newLen, rem);
                        cout << pos << "\n";
                    }
                    // File needs to be recreated
                    else {
                        fileDelete(tree, b);
                        cout << fileCreate(tree, b, newLen) << "\n";
                    }
                }
            }
            files[b].second = newLen;
        }
        // To also pass subtask 1
        else cout << "0\n";
    }
}
```

A: Drive Subtask 5 C++ Solution

```
#include <bits/stdc++.h>
using namespace std;
typedef long long ll;
typedef pair<int, int> ii;
```

```
int N, M;
unordered_map<int, ii> files;


struct node {
    ii rv;
    ii mv;
};

ii combRV(const ii& a, const ii& b) {
    return b.first ? b : a;
}

ii combMV(const ii& a, const ii& b) {
    return a.first >= b.first ? a : b;
}

node combine(const node& a, const node& b) {
    return {combRV(a.rv, b.rv), combMV(a.mv, b.mv)};
}

struct segtree {
    vector<node> nodes;
    unordered_set<int> hasv;

    segtree(int s): nodes(s * 4) {}

    void update(int n, int tl, int tr, int pos, int val) {
        if (tl == tr) {
            nodes[n] = {{val, tl}, {val, tl}};
            if (val) hasv.insert(tl);
            else hasv.erase(tl);
            return;
        }
        int tm = (tl + tr) / 2;
        if (pos <= tm) update(n * 2, tl, tm, pos, val);
        else update(n * 2 + 1, tm + 1, tr, pos, val);
        nodes[n] = combine(nodes[n * 2], nodes[n * 2 + 1]);
    }

    void clear() {
        vector<int> val(hasv.begin(), hasv.end());
        for (int a : val) update(1, 1, M, a, 0);
    }

    ii queryR(int n, int tl, int tr, int l, int r) {
        if (l <= tl && tr <= r) return nodes[n].rv;
        int tm = (tl + tr) / 2;
        ii rl{};
        ii rr{};
        if (l <= tm) rl = queryR(n * 2, tl, tm, l, r);
        if (tm < r) rr = queryR(n * 2 + 1, tm + 1, tr, l, r);
        return combRV(rl, rr);
    }
```

```
    ii getLoc(int n, int tl, int tr, int size) {
        if (tl == tr) return nodes[n].mv;
        int tm = (tl + tr) / 2;
        return nodes[n * 2].mv.first >= size ? getLoc(n * 2, tl, tm, size) : getLoc(n
* 2 + 1, tm + 1, tr, size);
    }
};

int fileCreate(segtree& tree, int fid, int size) {
    ii frr = tree.getLoc(1, 1, M, size);
    int loc = frr.second;
    files[fid] = {loc, size};
    tree.update(1, 1, M, loc, 0);
    int rem = frr.first - size;
    if (rem) tree.update(1, 1, M, loc + size, rem);
    return loc;
}

int fileDelete(segtree& tree, int fid) {
    int loc = files[fid].first;
    int len = files[fid].second;
    files.erase(fid);
    ii vp = tree.queryR(1, 1, M, 1, loc);
    ii vn{};
    if (loc + len <= M) vn = tree.queryR(1, 1, M, loc + len, loc + len);

    if (vn.first) {
        tree.update(1, 1, M, loc + len, 0);
        len += vn.first;
    }

    if (vp.second + vp.first == loc) tree.update(1, 1, M, vp.second, vp.first + len);
    else tree.update(1, 1, M, loc, len);
    return loc;
}

int getDiskSize(ll a) { return (a + 1474560 - 1) / 1474560; }

int main() {
    cin.tie(0)->sync_with_stdio(false);
    cin >> N >> M;

    int fdx = 1;

    segtree tree(M);
    tree.update(1, 1, M, 1, M);

    bool optimised = true;

    char a;
    int b;
    ll c;
    while (N--) {
        cin >> a;
        if (a == 'C') {
            cin >> c;
```

```
        cout << fileCreate(tree, fdx++, getDiskSize(c)) << "\n";
    }
    else if (a == 'D') {
        optimised = false;
        cin >> b;
        cout << fileDelete(tree, b) << "\n";
    }
    else if (a == 'M') {
        cin >> b >> c;
        int nsize = getDiskSize(c);
        int loc = files[b].first;
        int len = files[b].second;

        if (len == nsize) cout << loc << "\n";
        else {
            optimised = false;
            ii nv{};
            int npos = loc + len;
            if (npos <= M) nv = tree.queryR(1, 1, M, npos, npos);
            if (nsize < len) {
                if (nv.first) tree.update(1, 1, M, npos, 0);
                tree.update(1, 1, M, loc + nsize, len + nv.first - nsize);
                cout << loc << "\n";
            }
            else {
                if (nsize - len <= nv.first) {
                    tree.update(1, 1, M, npos, 0);
                    int rem = len + nv.first - nsize;
                    if (rem) tree.update(1, 1, M, loc + nsize, rem);
                    cout << loc << "\n";
                }
                else {
                    fileDelete(tree, b);
                    cout << fileCreate(tree, b, nsize) << "\n";
                }
            }
        }
        files[b].second = nsize;
    }
    else {
        if (optimised || files.empty()) {
            cout << "0\n";
            continue;
        }
        optimised = true;
        tree.clear();
        vector<ii> fll;
        for (auto a : files) fll.push_back({a.second.first, a.first});
        sort(fll.begin(), fll.end());
        int last = files[fll.back().second].first;
        int end = 1;
        for (auto a : fll) {
            files[a.second].first = end;
            end += files[a.second].second;
        }
        if (end <= M) tree.update(1, 1, M, end, M - end + 1);
```

```
            cout << last - files[fll.back().second].first << "\n";
        }
    }
}
```

A: Drive Full C++ Solution

```
#include <bits/stdc++.h>
using namespace std;
typedef long long ll;
#define fetchnull(n, var, def) (n ? n->var : def)

struct Node {
    Node* p;
    Node* l{};
    Node* r{};
    int height=1;
    int pos;
    int size;
    int mostl;
    int mostr;
    int longest=0;
    int tot;
    bool compressed=false;

    Node(Node* _p, int _pos, int _size) : p{_p}, pos{_pos}, size{_size}, mostl{_pos},
mostr{_pos + _size - 1}, tot{_size} {}
    void compress(int left) {
        pos = left + fetchnull(l, tot, 0);
        mostl = left;
        mostr = left + tot - 1;
        longest = 0;
        compressed = true;
    }
    void decompress() {
        if (!compressed) return;
        if (l) l->compress(mostl);
        if (r) r->compress(pos + size);
        compressed = false;
    }
    void update() {
        height = 1 + max(fetchnull(l, height, 0), fetchnull(r, height, 0));
        mostl = fetchnull(l, mostl, pos);
        mostr = fetchnull(r, mostr, pos + size - 1);
        longest = max(fetchnull(l, longest, 0), fetchnull(r, longest, 0));
        if (l) longest = max(longest, pos - l->mostr - 1);
        if (r) longest = max(longest, r->mostl - pos - size);
        tot = size + fetchnull(l, tot, 0) + fetchnull(r, tot, 0);
    }
};

int N, M;

void rotL(Node*& x) {
    x->decompress();
    x->r->decompress();
    Node* nx = x->r;
```

```
    nx->p = x->p;
    x->r = nx->l;
    if (x->r) x->r->p = x;
    nx->l = x;
    x->p = nx;
    x->update();
    nx->update();
    x = nx;
}
void rotR(Node*& x) {
    x->decompress();
    x->l->decompress();
    Node* nx = x->l;
    nx->p = x->p;
    x->l = nx->r;
    if (x->l) x->l->p = x;
    nx->r = x;
    x->p = nx;
    x->update();
    nx->update();
    x = nx;
}

int balance(Node* n) { return n ? fetchnull(n->l, height, 0) - fetchnull(n->r, height,
0) : 0; }
void updateRebalance(Node*& n) {
    n->decompress();
    n->update();
    int bal = balance(n);
    if (bal > 1) {
        if (balance(n->l) < 0) rotL(n->l);
        rotR(n);
    }
    else if (bal < -1) {
        if (balance(n->r) > 0) rotR(n->r);
        rotL(n);
    }
}

Node* createFile(Node*& n, Node* p, int tl, int tr, int size) {
    if (!n) return n = new Node(p, tl, size);
    n->decompress();
    int leftfree = n->mostl - tl;
    if (n->l) leftfree = max({leftfree, n->l->longest, n->pos - n->l->mostr - 1});
     Node* res = (size <= leftfree) ? createFile(n->l, n, tl, n->pos - 1, size) :
createFile(n->r, n, n->pos + n->size, tr, size);
    updateRebalance(n);
    return res;
}

bool tryModifyFile(Node*& n, int tr, int pos, int size) {
    bool res = true;
    if (pos < n->pos) res = tryModifyFile (n->l, n->pos - 1, pos, size);
    else if (pos > n->pos) res = tryModifyFile(n->r, tr, pos, size);
    else if (size <= (n->r ? n->r->mostl : tr + 1) - n->pos) {
        n->size = size;
```

```
        res = false;
    }
    n->update();
    return res;
}

Node* fetchLeft(Node*& n) {
    n->decompress();
    if (!n->l) {
        Node* tmp = n;
        if (n->r) n->r->p = n->p;
        n = n->r;
        return tmp;
    }
    Node* res = fetchLeft(n->l);
    updateRebalance(n);
    return res;
}
void deleteFile(Node*& n, int pos) {
    if (pos < n->pos) deleteFile(n->l, pos);
    else if (pos > n->pos) deleteFile(n->r, pos);
    else {
        Node* newNode;
        if (!n->l) {
            newNode = n->r;
            if (newNode) newNode->p = n->p;
        }
        else if (!n->r) {
            newNode = n->l;
            if (newNode) newNode->p = n->p;
        }
        else {
            newNode = fetchLeft(n->r);
            newNode->p = n->p;
            newNode->l = n->l;
            if (newNode->l) newNode->l->p = newNode;
            newNode->r = n->r;
            if (newNode->r) newNode->r->p = newNode;
        }
        delete n;
        n = newNode;
        if (!n) return;
    }
    updateRebalance(n);
}

void decompressAncestors(Node* n) {
    if (n->p) decompressAncestors(n->p);
    n->decompress();
}

int getDiskSize(ll a) { return (a + 1474560 - 1) / 1474560; }
int main() {
    cin.tie(0);
    ios_base::sync_with_stdio(false);
    cin >> N >> M;
```

```
    Node* root{};
    unordered_map<int, Node*> files;
    int nextFileID = 1;
    char a;
    int b;
    ll c;
    while (N--) {
        cin >> a;
        if (a == 'C') {
            cin >> c;
          cout << (files[nextFileID++] = createFile(root, nullptr, 1, M, getDiskSize(c)))-
>pos << "\n";
        }
        else if (a == 'D') {
            cin >> b;
            decompressAncestors(files[b]);
            cout << files[b]->pos << "\n";
            deleteFile(root, files[b]->pos);
            files.erase(b);
        }
        else if (a == 'M') {
            cin >> b >> c;
            int size = getDiskSize(c);
            decompressAncestors(files[b]);
            if (tryModifyFile(root, M, files[b]->pos, size)) {
                deleteFile(root, files[b]->pos);
                files[b] = createFile(root, nullptr, 1, M, size);
            }
            cout << files[b]->pos << "\n";
        }
        else {
            if (!root) cout << "0\n";
            else {
                cout << root->mostr - root->tot << "\n";
                root->compress(1);
            }
        }
    }
}
```