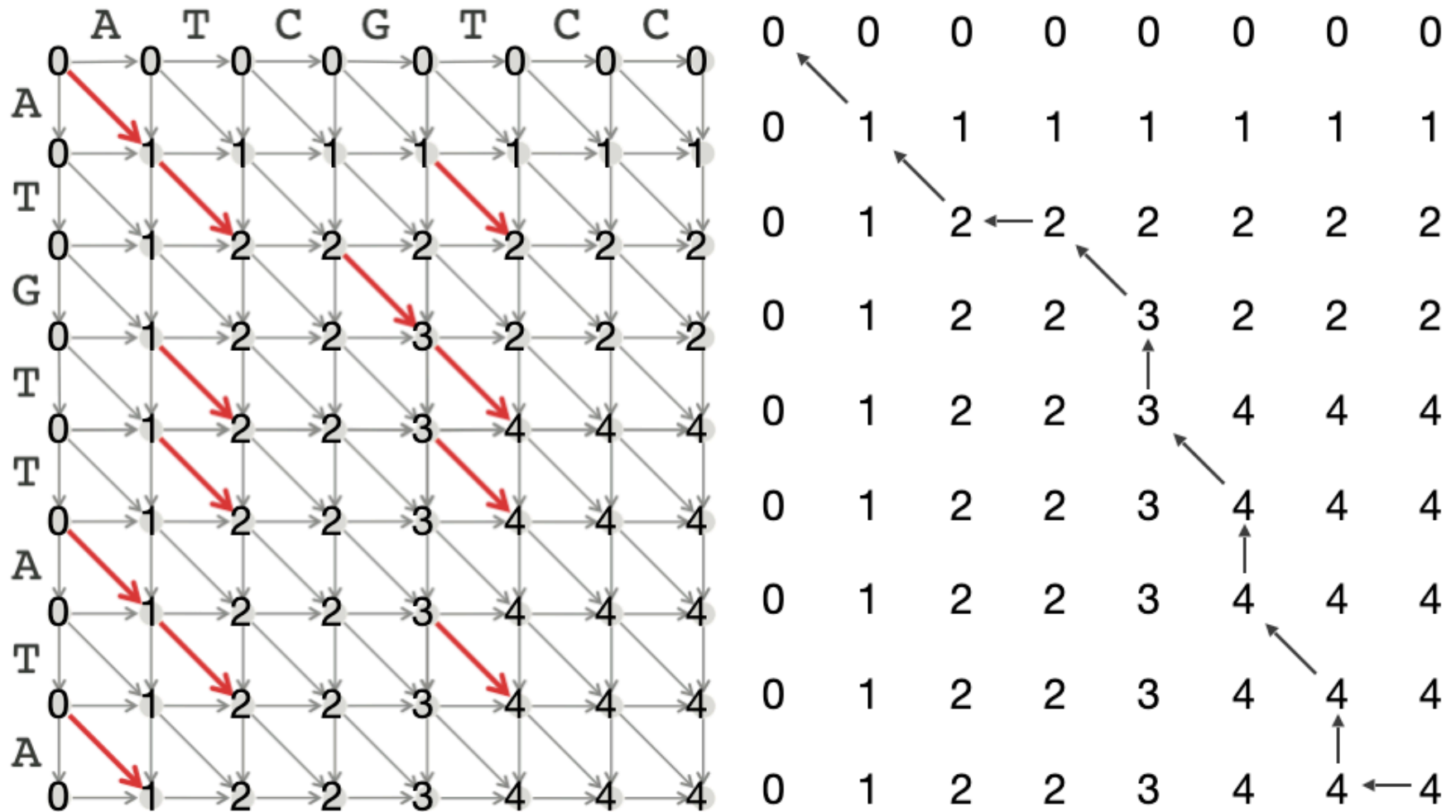


2D and 3D optimization: edit distance

- UVa 13146
- **Problem:** transform one string into another using as few edits as possible.
 - An edit inserts, deletes, or substitutes one character; a match costs nothing.
 - **Example:** kitten → sitten → sittin → sitting : 3 edits.
 - **DP state:** $D(i, j)$ is the minimum cost to transform the first i characters of one string into the first j of the other.

2D and 3D optimization: edit distance

- **Problem:** transform one string into another using as few edits as possible.



2D and 3D optimization: edit distance

Recurrence:

$$D_{i,j} = \max \begin{cases} D_{i-1,j-1} + s_{x_i,y_j} & \text{match/mismatch} \\ D_{i-1,j} + g & \text{skip } x_i \text{ (gap)} \\ D_{i,j-1} + g & \text{skip } y_j \text{ (gap)} \end{cases}$$

```
x = "intention"
y = "intuitions"
D = [[i for i in range(len(y) + 1)]]
for xi in range(len(x)):
    D.append([0 for _ in range(len(y) + 1)])
    D[-1][0] = xi + 1
    for yi in range(len(y)):
        D[xi + 1][yi + 1] = min(
            D[xi][yi + 1] + 1, # delete in x
            D[xi + 1][yi] + 1, # delete in y
            D[xi][yi] + (1 if y[yi] != x[xi] else 0), # match/mismatch
        )
score = D[len(x)][len(y)]
```

Edit distance: backtracking

```
nx, ny = '', ''
xi, yi = len(x), len(y)
while xi > 0 or yi > 0:
    if xi > 0 and yi > 0 and D[xi][yi] == D[xi - 1][yi - 1] + (1 if x[xi - 1] != y[yi - 1] else 0):
        nx = x[xi - 1] + nx
        ny = y[yi - 1] + ny
        xi -= 1
        yi -= 1
    elif yi > 0 and D[xi][yi] == D[xi][yi - 1] + 1:
        nx = "-" + nx
        ny = y[yi - 1] + ny
        yi -= 1
    else:
        nx = x[xi - 1] + nx
        ny = "-" + ny
        xi -= 1

print(score)
print(nx)
print(''.join(' ' if a == b else '|' for a, b in zip(nx, ny)))
print(ny)
# 3
# intention-
#   ||   |
# intuitions
```

2025 Div1/Div2: Shh

- <http://ml3490204.vicp.cc:38741/d/luogu/p/P17389>
- Given a string, compute the minimum number of characters to change such that it has exactly k instances of the substring shh. Furthermore, compute the number of distinct strings with exactly k instances of shh that can be obtained by changing exactly that many characters
- Example:

Input:

3 0

shh

Output:

1 8

- Changing 1 letter removes all shh's
- Each letter can be changed to 25 other letters, so $3 \times 25 = 75 \mod 67 = 8$

2025 Div1/Div2: Shh

- Brute force: too slow
- Idea: build string by adding one character at time
 - **DP state:** a triple
 - Length of prefix
 - How many shh's we did thus far? (k at most)
 - Is our string ending in a prefix of shh? (3 options)
 - For each state, we store a pair: min number of changes, how many ways
- Complexity: $O(26 \times 3 \times nk) = O(nk) \approx O(n^2)$ (n < 67 so super fast)

```
dp = [[[INF, 0] for _ in range(3)] for _ in range(k + 1)]
dp[0][0] = (0, 1)
for c in s:
    # Use this to roll old result into a new!
    ndp = [[[INF, 0] for _ in range(3)] for _ in range(k + 1)]
    for cnt in range(k + 1):
        for st in range(3):
            cost, ways = dp[cnt][st]
            if cost == INF: continue
            # Try replacing current character with any letter
            for rep in "abcdefghijklmnopqrstuvwxyz":
                new_cost = cost + (rep != c)

                # Update state
                if rep == 's': new_st = 1 # s
                elif rep == 'h' and st == 1: new_st = 2 # sh
                elif rep == 'h' and st == 2: new_st = 0 # shh
                else: new_st = 0
                new_cnt = cnt + (st == 2 and rep == 'h')
                if new_cnt > k: continue

                old_cost, old_ways = ndp[new_cnt][new_st]
                if new_cost < old_cost:
                    ndp[new_cnt][new_st] = (new_cost, ways)
                elif new_cost == old_cost:
                    ndp[new_cnt][new_st] = (
                        old_cost, (old_ways + ways) % MOD
                    )

    dp = ndp
best = min(dp[k][state][0] for state in range(3))
ways = 0
for state in range(3):
    if dp[k][state][0] == best:
        ways = (ways + dp[k][state][1]) % MOD
```

More problems...

- **UVa 10029 — Edit Step Ladders**: find the longest lexicographically increasing chain of dictionary words; neighbors differ by one insertion, deletion, or substitution.
 - DP state: longest valid chain ending at a word. Only earlier words can precede it.

1D optimization: LIS

- **LIS problem:** keep the **longest strictly increasing subsequence**, preserving order but allowing skipped elements.
 - **Example:** [3, 1, 2, 5, 4] has [1, 2, 5] .
- **DP state:** L_i is the longest increasing subsequence ending at index i .

```
arr = [3, 1, 2, 5, 4]
lis = [1] * len(arr)
for i in range(1, len(arr)):
    for prev in range(0, i):
        if arr[i] > arr[prev]:
            lis[i] = max(lis[i], lis[prev] + 1)
print(max(lis))
```

- Complexity: $O(n^2)$

LIS optimization: binary search

- Keep maintaining the minimum last element of increasing sequence of size i
 - They must be sorted!
 - When adding new element, find largest i so that new element can extend it!
 - Can use binary search here because list is sorted

```
arr = [3, 1, 2, 5, 4]
ans = [arr[0]]
for i in range(1, len(arr)):
    if arr[i] > ans[-1]:
        ans.append(arr[i])
    else:
        # Use bisect: ans[bisect.bisect_left(ans, arr[i])] = arr[i]
        # Or do binary search yourself:
        lo, hi = 0, len(ans) - 1
        while lo < hi:
            mid = lo + (hi - lo) // 2
            if ans[mid] < arr[i]:
                lo = mid + 1
            else:
                hi = mid
        ans[lo] = arr[i]
print(len(ans))
```

2024 PacNW Div2: Homework Help

- There is a secret permutation of n integers. In a single query, you can request the number of inversions of a given subarray. Given the ability to make n queries, compute the length of the longest increasing subsequence of the array.
- **Interactive problem!**

2024 PacNW Div2: Homework Help

- There is a secret permutation of n integers. In a single query, you can request the number of inversions of a given subarray. Given the ability to make n queries, compute the length of the longest increasing subsequence of the array.
- **Interactive problem!**
- **Adhoc + dynamic programming**
 - We are not rate-limited, so we can ask about the number of inversions for every prefix
 - Then we can reconstruct the array
- **Solution:**
 - Once array a is constructed, we can get the length of the longest LIS in any of its subarrays by:
$$LIS(i) = \max_{j:j < i \text{ and } a_j < a_i} \{LIS(j) + 1\}$$
 - $LIS(i)$ calculates the LIS of $a[1, 2, \dots, i]$ (i.e., `a[:i+1]`)
 - $LIS(1) = 1$, obviously
 - Main trick: construction of a

2024 PacNW Div2: Homework Help

- Recover array (example: `a = [3, 1, 4, 2]`)
 - `? 1 1` (array is `[3]`) $\xrightarrow{\text{SOON}}$ answer is 0
 - `? 1 2` (array is `[3,1]`) $\xrightarrow{\text{SOON}}$ answer is 1
 - `? 1 3` (array is `[3,1,4]`) $\xrightarrow{\text{SOON}}$ answer is 1
 - `? 1 4` (array is `[3,1,4,2]`) $\xrightarrow{\text{SOON}}$ answer is 3
- Do LIS

```
n = int(input())
# Inversion counts of every prefix
inv = [0] * n
for i in range(1, n):
    print("?", 1, i + 1, flush=True)
    if (inv[i] := int(input())) == -1: sys.exit()
# Reconstruct the permutation
order = [] # order stores original indices sorted by hidden values
for i in range(n):
    diff = inv[i] - (inv[i - 1] if i else 0)
    order.insert(i - diff, i) # move position of current element where it belongs
a = [0] * n
for rank, idx in enumerate(order):
    a[idx] = rank + 1
# Standard O(n^2) LIS DP
dp = [1] * n
for i in range(n):
    for j in range(i):
        if a[j] < a[i]:
```

Back to Edit

- **UVa 12747 — Back to Edit:** make two permutations identical with the fewest insertions/deletions; substitutions are forbidden.
 - Find a longest common subsequence (LCS) and its length L ; then the answer is $2(N - L)$
- Example:
 - $A = [1, 3, 5, 4, 2]$
 - $B = [1, 5, 4, 3, 2]$
- LCS is $O(N^2)$ (same as edit distance), but these are permutations, so...
 - Now replace each number in A with its index in B !
 - Do $O(N \log N)$ LIS on that sequence

