19 (100 PTS.) Self edit distance (take 2).

You are given a string $S = s_1 s_2 \dots s_n$ of length n over a finite alphabet Σ. You are also given a budget $k \leq \lceil n/2 \rceil$. Describe an algorithm, as fast as possible, that computes two subsequences U, V of S, such that (i) the edit distance between U and V is at most k (under the restriction specified below), and (ii) the total length $|U| + |V|$ is maximized. Here, every edit operation has cost 1. For example, for `urbana-banananan` with a budget $k = 2$, a solution might be the subsequences

u*rbana*-*banananan*.

As the edit distance between *rbananana* and *bananan* is two. The value of this solution is 16. (This is not the optimal solution in this case.)

Importantly, the two subsequences are not necessarily disjoint, but you are not allowed to match an original character to itself in the edit distance (thus, you can not just match the whole input string to itself).

Solution:

In the following, let $[\![\text{bla}]\!]$ be 1 if bla is true, and 0 otherwise. For $i \leq j$, let $f(i, j, \alpha)$ be the maximal total length of two subsequences of $s_1 \dots s_t$, with edit distance at most α, where i is the location where one of the strings ends, and the j is the location where the other string ends (all under the constraint of not matching an original character to itself).

Let first deal with some easy boundary cases:

(i) $\alpha < 0 \implies f(i, j, \alpha) = -\infty$.
(ii) $i = 0$ and $j = 0 \implies f(i, j, \alpha) = 0$.
(iii) $i = 0 \implies f(i, j, \alpha) = \min(\alpha, j)$.
(iv) $j = 0 \implies f(i, j, \alpha) = \min(\alpha, i)$.

Now we have to go through the standard edit distance alignment options. First, we might just decide to skip the ith letter, and $f(i, j, \alpha)$ would be equal to $f(i-1, j, \alpha)$. Similarly, $f(i, j, \alpha)$ might be equal to $f(i, j-1, \alpha)$.

If $i \neq j$, and we align s_i and s_j, then the value of the solution is

$$\beta_1 = \mathbf{2} + f\Big(i-1, j-1, \alpha - [\![s_i \neq s_j]\!]\Big).$$

The 2 is here because both computed subsequences get longer each by one character.

If we delete s_i, then $\beta_2 = 1 + f\Big(i-1, j, \alpha - 1\Big)$.

If we delete s_j, then $\beta_3 = 1 + f\Big(i, j-1, alpha - 1\Big)$.

Putting everything together, assuming we are not in a boundary case, we have

$$f(i, j, \alpha) = \max \begin{cases} f(i-1, j, \alpha) & \texttt{// Skip } i\texttt{th character} \\ f(i, j-1, \alpha) & \texttt{// Skip } j\texttt{th character} \\ 1 + f(i-1, j, \alpha - 1) & \texttt{// Delete } i\texttt{th character} \\ 1 + f(i, j-1, \alpha - 1) & \texttt{// Insert } j\texttt{th character} \\ \mathbf{2} + f\Big(i-1, j-1, \alpha - [\![s_i \neq s_j]\!]\Big) & i \neq j \end{cases}$$

The correctness follows directly from the standard edit distance argumentation. This recursive function has 3 parameters the first two can have $\neq n+1$ values, and the third one can be $k+2$ values. We conclude that the number of distinct calls to f is $O(n^2k)$. Since each recursive evaluation requires $O(1)$ different recursive evaluations, it follows that turning this function into a recursive function, an then using memoization, results in an algorithm with running time $O(n^2k)$. Here, the desired final result is $f(n,n,k)$.

20 (100 PTS.) Trees have needs, but then who doesn't?

We are given a tree $T = (V, E)$ with n vertices. Assume that the degree of all the vertices in T is at most 3. You are given a function $f : V \to \{0, 1, 2, 3\}$. The task is to compute a subset X of edges, such that for every node $v \in V$, there are at least $f(v)$ distinct edges in X that are adjacent to v.

Describe an algorithm, as fast as possible, that computes the minimum size set $X \subseteq E$ that meets the needs of all the nodes in the tree. The algorithm should output both $|X|$ and X itself.

Solution:

We root the tree at an arbitrary vertex r that is either of degree two or one (this takes $O(n)$ time to do) – since every tree has a leaf, this is always possible. Now, every vertex has pointers left(v), right(v), which are nil if these children nodes do not exist.

In the following, let [condition] be the function that is 1 if the condition is true, and 0 otherwise. The recursive algorithm is the following, where we call **cover**$(r, 0)$.

A call to **cover**(v, s) returns the cheapest feasible solution that meets all the demands in the subtree of v, where $s \in \{0, 1\}$ indicates whether or not the edge to the parent is included in the solution (and needed to be paid for.

```
cover($v, s$):
    $b \leftarrow +\infty$, $c \leftarrow \langle 0, 0 \rangle$
    for $\ell = 0, 1$ do
        for $r = 0, 1$ do
            $\Delta \leftarrow s + [\text{left}(v) \neq \text{nil}] \cdot \ell + [\text{right}(v) \neq \text{nil}] \cdot r$
            if $\Delta \geq f(v)$ then
                $p_{\text{left}} \leftarrow 0$, $p_{\text{right}} \leftarrow 0$
                if $\text{left}(v) \neq \text{nil}$ then
                    $p_{\text{left}} \leftarrow$ cover$(\text{left}(v), \ell)$
                if $\text{right}(v) \neq \text{nil}$ then
                    $p_{\text{right}} \leftarrow$ cover$(\text{right}(v), r)$
                $\alpha \leftarrow p_{\text{left}} + p_{\text{right}} + s$
                if $\alpha < b$ then
                    $b \leftarrow \alpha$
                    $c \leftarrow \langle \ell, r \rangle$

    $\text{sol}[v, s] \leftarrow c$        // For printing the solution later
    return $b$
```

There are $2n$ distinct recursive calls, so memoization readily yields $O(n)$ time algorithm, since the recursive function takes $O(1)$ time ignoring the recursive calls.

Printing the solution is now easy. Indeed, all we have to do is to first call the above procedure. Next, we call **printSolution**(nil, r, 0), where:

```
// $p$:  Parent of current node $v$
printSolution($p, v, s$):
    if $v =$ nil then
        return
    if $s = 1$ then
        print "Edge $pv$ in optimal solution"
    $c \leftarrow \mathrm{sol}[v, s]$
    $\langle \ell, r \rangle \leftarrow c$
    if $\mathrm{left}(v) \neq$ nil then
        printSolution($v, \mathrm{left}(v), \ell$)
    if $\mathrm{right}(v) \neq$ nil then
        printSolution($v, \mathrm{right}(v), r$)
```

21 (100 PTS.) Trust but shadow.

A classical tactic in war, is for a small force to "shadow" a bigger force of the enemy. So, assume we have a main army moving along a sequence $p_1, \ldots, p_n$ of locations. Initially, the army start at p_1, and every day it moves forward to the next location, spending the night there. The shadow army, similarly, knowing their enemy path, has a sequence of locations $q_1, q_2, \ldots, q_m$ that it is planning to travel through.

The shadow army, being much smaller, can move much faster than the main army. In particular, it can move through as many locations as it wants in one day.

The important thing with shadowing, is that the shadow army should not be too close to the main army when they both camp at night (because that would trigger a battle, which would be bad). Similarly, the shadow army should not be too far from the main army, as then it can not keep track of it.

The task at hand is to come up with a schedule for the shadow army. In the beginning of ith day, the main army is at location p_i, and the shadow army is at location $q_{\pi(i)}$ (π is what you have to compute). We require that $\pi(1) = 1$, $\pi(i+1) \geq \pi(i)$, and $\pi(n) = m$. The locations $p_1, \ldots, p_n$ and $q_1, \ldots, q_m$ are points in the plane (and are the input to the algorithm), and the distance between two locations is the Euclidean distance between them.

21.A. (40 PTS.) An interval $[x, y] \subseteq \mathbb{R}$ is ***feasible*** if there exists a valid schedule π, such that, for all i, we have $\|p_i - q_{\pi(i)}\| \in [x, y]$. Given such an interval $[x, y]$, describe a dynamic program algorithm, as fast as possible, that uses as little space as possible, that decides if $[x, y]$ is feasible (no need to output the schedule).

Solution:

Let $f(i, j)$ be a boolean function that is TRUE, if there is a feasible schedule that ends up with the main army being at p_i and the shadow army at location q_j. We have

$$f(i,j) = \begin{cases} \text{FALSE} & \|p_1 - q_1\| \notin [x,y] \\ \text{FALSE} & \|p_i - q_j\| \notin [x,y] \\ \text{TRUE} & i = 1 \text{ and } j = 1 \\ \bigvee_{k=1,\ldots,j} f(i-1,k) & \text{otherwise.} \qquad // \ \|p_i - q_j\| \in [x,y]. \end{cases}$$

The correctness of this recurrence is hopefully clear – consider the last configuration before (p_i, q_j). It must be some configuration (p_{i-1}, q_ℓ), for $\ell \leq j$, such that (p_{i-1}, q_ℓ) is reachable, and $\|p_i - q_j\| \in [x, y]$.

Turning this into a recursive algorithm, this recursion has $O(n^2)$ distinct calls, and each one takes $O(n)$ time to compute from previous values, so overall the running time is $O(n^3)$. However, one can do better with a bit of cleverness. We restate the recurrence using an external function:

$$g(i,j) = \begin{cases} \text{FALSE} & \|p_1 - q_1\| \notin [x,y] \\ \text{FALSE} & \|p_i - q_j\| \notin [x,y] \\ \text{TRUE} & i = 1 \text{ and } j = 1 \\ G(i-1,j) & \text{otherwise.} \qquad // \ \|p_i - q_j\| \in [x,y]. \end{cases}$$

$$G(i-1,j) = G(i-1,j-1) \vee g(i-1,j),$$

where $G(0,j) =$ TRUE and $G(i,0) =$ FALSE. Using easy induction, one can show that $G(i-1,j) = \bigvee_{k=1,\ldots,j} f(i-1,k)$ and $g(i,j) = f(i,j)$. Memoization would give us running time $O(nm)$ and space $O(nm)$. But one can do better – we get the following dynamic program:

```
DP($p_1, \ldots, p_n$, $q_1, \ldots, q_m, x, y$):
    If $\|p_1 - q_1\| \notin [x,y]$ then return FALSE
    $G_n[0\ldots,m], g[0\ldots,m]$:  Allocate arrays
    $G_o[0\ldots,m] \leftarrow$ TRUE
    for $i = 1, \ldots, n$ do
        for $j = 1, \ldots, m$ do
            if $i = 1$ and $j = 1$ then
                $g_n[j] \leftarrow$ TRUE
                continue
            if $\|p_i - q_j\| \notin [x,y]$ then
                $g_n[j] \leftarrow$ FALSE
                continue
            $g_n[j] \leftarrow G_o[j]$
        end for
        $G_n[1] \leftarrow g_n[1]$
        for $j = 2, \ldots, m$ do
            $G_n[j] \leftarrow G_n[j-1] \vee g_n[j]$
        $G_o \leftarrow G_n$
    return $G_o[m]$
```

The running time of this algorithm is $O(nm)$, and the space it uses is $O(n+m)$. The code can be modified to use only one array instead of G_n and G_o, but hopefully this version is clearer.

Rubric: Running time $O(nm^2)$ or (n^2m) is worth 30 points (assuming everything else is done correctly).

Rubric: Running time $O(nm)$ is worth 40 points (assuming everything else is done correctly).

Solution:

A more efficient solution. Let $f(i)$ be the minimal location the shadow army can be in the end of the ith day. Observe that either $f(i) = f(i-1)$, or alternatively, you need to compute the minimal $j > f(i-1)$, such that $\|p_i - q_j\| \in [x,y]$. Here, we require that $f(1) = 1$, and $f(n) = m$.

Lemma 7.1. *The above greedy strategy works, and generates a valid schedule.*

Proof: A solution here is a subsequence $1 = i_1 \leq i_2 \leq \cdots \leq i_n = m$. Given an optimal solution $o_1, o_2, \ldots, o_n$, consider the first index where $i_k < o_k$, and observe that change o_k to be i_k preserve the schedule being valid. As such, repeating this rewriting, results in a new schedule that is feasible, and is equal to the greedy schedule. ■

The resulting running time is $O(n+m)$, as the algorithm only inspects $O(n+m)$ pairs of points during this process.

The only reason the greedy solution works here is because the problem was a bit over simplified. If one requires that all the middle points used by the shadow army are within the allowable distance, then the greedy strategy fails. For people interested in the original problem, this problem is a variant of the Fréchet distance between curves.

Rubric: Without proof of correctness, this solution is worth 10 points. With proof correctness, it is worth 40 points.

21.B. (30 PTS.) Describe an algorithm, as fast as possible, that computes the maximal x, such that the interval $[x, \infty]$ is feasible.

Solution:

Compute the set of all distances $X = \{\|p_i - q_j\| \mid i = 1, \ldots, n, j = 1, \ldots, m\}$. The desired $x \in X$, and as such we can just do binary search for this formally. Formally, sort X, and let $x_1 \leq x_2 \ldots \leq x_{nm}$ be the values in sorted order.

Conceptually, let

$$f[i] \leftarrow \mathbf{DP}(p_1, \ldots, p_n,\ q_1, \ldots, q_m, x_i, +\infty),$$

for $i = 1, \ldots, nm$. Here **DP** is the algorithm from the previous part. Observe that the array $f[1 \ldots mn]$ has an index t, such that $f[1 \ldots t]$ is all TRUE, and $f[t+1 \ldots nm]$ is FALSE. Our task is to compute this index t, and $[x_t, +\infty]$ is the desired interval. This is of course, no more than a binary search, and here is the resulting algorithm.

```
FindMF$(i, j)$:
    if DP$(p_1, \ldots, p_n,\ q_1, \ldots, q_m, x_j, +\infty)$ then
        return $x_j$
    $\ell = \lfloor (i+j)/2 \rfloor$
    $f \leftarrow$ DP$(p_1, \ldots, p_n,\ q_1, \ldots, q_m, x_\ell, +\infty)$
    if $f$ is TRUE then
        return FindMF$(\ell, j-1)$
    return FindMF$(i, \ell-1)$
```

The main procedure is the following:

```
main$(p_1, \ldots, p_n,\ q_1, \ldots, q_m)$:
    $X = \{\|p_i - q_j\| \mid i = 1, \ldots, n, j = 1, \ldots, m\}$
    Sort $X$ // $O(mn \log(mn))$ time.   $X = \langle x_1, x_2, \ldots, x_{mn} \rangle$
    return FindMF$(1, nm)$
```

Sorting costs $O(nm \log(nm))$ time, and the binary search performs $O(\log(nm))$ calls to **DP**. Each such call takes $O(nm)$ time. So overall, the running time is $O(nm \log(nm))$.

21.C. (30 PTS.) The ***instability*** of an interval $[x, y]$, with $0 < x < y$, is the ratio y/x. Describe an algorithm, as fast as possible, that computes the interval with minimum instability, among all feasible intervals.

Solution:

Rubric: The basic solution is 20 points.

As above, compute the set $X = \{\|p_i - q_j\| \mid i = 1, \ldots, n, j = 1, \ldots, m\}$, and sort it. For any pair of indices $1 \leq i \leq j \leq nm$, decide if $[x_i, x_j]$ is feasible using **DP**. Among all feasible intervals $[x, y]$ computed, return the one that minimizes y/x. The running time is $O(n^3m^3)$.

Rubric: The following more clever solution is 30 points.

A more clever solution for every $x_i \in X$, it computes the minimum $x_j \geq x_i$ such that $[x_i, x_j]$ is feasible. This can be done in $O(nm \log(nm))$ time, using binary search as was done in the above (B) part. This takes $O((nm)(nm \log nm)) = O(n^2m^2 \log(nm))$ time overall. Now, output the best feasible interval computed. The overall running time is $O(n^2m^2 \log(nm))$.

Using the improved solution for part (A), leads to $O(nm(n + m) \log nm))$ running time. Getting a better running time is probably not trivial.