Notes for Lecture 13

1 Edit Distance

1.1 Definition

When you run a spell checker on a text, and it finds a word not in the dictionary, it normally proposes a choice of possible corrections.

If it finds `stell` it might suggest `tell`, `swell`, `stull`, `still`, `steel`, `steal`, `stall`, `spell`, `smell`, `shell`, and `sell`.

As part of the heuristic used to propose alternatives, words that are "close" to the misspelled word are proposed. We will now see a formal definition of "distance" between strings, and a simple but efficient algorithm to compute such distance.

The distance between two strings $x = x_1 \cdots x_n$ and $y = y_1 \cdots y_m$ is the minimum number of "errors" (edit operations) needed to transform x into y, where possible operations are:

- insert a character.
  $insert(x, i, a) = x_1 x_2 \cdots x_i a x_{i+1} \cdots x_n$.
- delete a character.
  $delete(x, i) = x_i x_2 \cdots x_{i-1} x_{i+1} \cdots x_n$.
- modify a character.
  $modify(x, i, a) = x_1 x_2 \cdots x_{i-1} a x_{i+1} \cdots x_n$.

For example, if $x = aabab$ and $y = babb$, then one 3-steps way to go from x to y is

a	a	b	a	b		x
b	a	a	b	a	b	x' = insert(x,0,b)
b	a	b	a	b		x" = delete (x',2)
b	a	b	b			y = delete (x",4)

another sequence (still in three steps) is

a	a	b	a	b	x
a	b	a	b		x' = delete (x,1)
b	a	b			x" = delete(x',1)
b	a	b	b		y = insert (x",3,b)

Can you do better?

1.2 Computing Edit Distance

To transform $x_1 \cdots x_n$ into $y_1 \cdots y_m$ we have three choices:

- put y_m at the end: $x \to x_1 \cdots x_n y_m$ and then transform $x_1 \cdots x_n$ into $y_1 \cdots y_{m-1}$.
- delete x_n: $x \to x_1 \cdots x_{n-1}$ and then transform $x_1 \cdots x_{n-1}$ into $y_1 \cdots y_m$.

- change x_n into y_m (if they are different): $x \to x_1 \cdots x_{n-1} y_m$ and then transform $x_1 \cdots x_{n-1}$ into $y_1 \cdots y_{m-1}$.

This suggests a recursive scheme where the sub-problems are of the form "how many operations do we need to transform $x_1 \cdots x_i$ into $y_1 \cdots y_j$.

Our dynamic programming solution will be to define a $(n+1) \times (m+1)$ matrix $M[\cdot,\cdot]$, that we will fill so that for every $0 \le i \le n$ and $0 \le j \le m$, $M[i,j]$ is the minimum number of operations to transform $x_1 \cdots x_i$ into $y_1 \cdots y_j$.

The content of our matrix M can be formalized recursively as follows:

- $M[0,j] = j$ because the only way to transform the empty string into $y_1 \cdots y_j$ is to add the j characters $y_1, \ldots, y_j$.
- $M[i,0] = i$ for similar reasons.
- For $i, j \ge 1$,

$$\begin{aligned} M[i,j] \quad = \min\{ \quad & M[i-1,j]+1, \\ & M[i,j-1]+1, \\ & M[i-1,j-1] + \mathit{change}(x_i,y_j)\} \end{aligned}$$

  where $\mathit{change}(x_i,y_j) = 1$ if $x_i \neq y_j$ and $\mathit{change}(x_i,y_j) = 0$ otherwise.

As an example, consider again $x = aabab$ and $y = babb$

	λ	b	a	b	b
λ	0	1	2	3	4
a	1	1	1	2	3
a	2	2	1	2	3
b	3	2	2	1	2
a	4	3	2	2	2
b	5	4	3	2	2

What is, then, the edit distance between x and y?

The table has $\Theta(nm)$ entries, each one computable in constant time. One can construct an auxiliary table $Op[\cdot,\cdot]$ such that $Op[\cdot,\cdot]$ specifies what is the first operation to do in order to optimally transform $x_1 \cdots x_i$ into $y_1 \cdots y_j$. The full algorithm that fills the matrices can be specified in a few lines

```
algorithm EdDist(x,y)
    n = length(x)
    m = length(y)
    for i = 0 to n
        M[i,0] = i
    for j = 0 to m
        M[0,j] = j
```

```
for i = 1 to n
  for j = 1 to m
    if x_i == y_j then change = 0 else change = 1
    M[i,j] = M[i-1,j] + 1;  Op[i,j] = delete(x,i)
    if M[i,j-1] + 1 < M[i,j] then
      M[i,j] = M[i,j-1] + 1;  Op[i,j] = insert(x,i,y_j)
    if M[i-1,j-1] + change < M[i,j] then
      M[i,j] = M[i-1,j-1] + change
      if (change == 0) then Op[i,j] = none
      else Op[i,j] = change(x,i,y_j)
```

2 Longest Common Subsequence

A *subsequence* of a string is obtained by taking a string and possibly deleting elements.

If $x_1 \cdots x_n$ is a string and $1 \leq i_1 < i_2 < \cdots < i_k \leq n$ is a strictly increasing sequence of indices, then $x_{i_1} x_{i_2} \cdots x_{i_k}$ is a subsequence of x. For example, **art** is a subsequence of **a**lgo**r**i**t**hm.

In the *longest common subsequence problem*, given strings x and y we want to find the longest string that is a subsequence of both.

For example, **art** is the longest common subsequence of **a**lgo**r**i**t**hm and p**ar**achu**t**e.

As usual, we need to find a recursive solution to our problem, and see how the problem on strings of a certain length can be reduced to the same problem on smaller strings.

The length of the l.c.s. of $x = x_1 \cdots x_n$ and $y = y_1 \cdots y_m$ is either

- The length of the l.c.s. of $x_1 \cdots x_{n-1}$ and $y_1 \cdots y_m$ or;
- The length of the l.c.s. of $x_1 \cdots x_n$ and $y_1 \cdots y_{m-1}$ or;
- 1 + the length of the l.c.s. of $x_1 \cdots x_{n-1}$ and $y_1 \cdots y_{m-1}$, if $x_n = y_m$.

The above observation shows that the computation of the length of the l.c.s. of x and y reduces to problems of the form "what is the length of the l.c.s. between $x_1 \cdots x_i$ and $y_1 \cdots y_i$?"

Our dynamic programming solution uses an $(n+1) \times (m+1)$ matrix M such that for every $0 \leq i \leq n$ and $0 \leq j \leq m$, $M[i,j]$ contains the length of the l.c.s. between $x_1 \cdots x_i$ and $y_1 \cdots y_j$. The matrix has the following formal recursive definition

- $M[i,0] = 0$
- $M[0,j] = 0$
- $$\begin{aligned} M[i,j] \quad = \max\{ \quad & M[i-1,j] \\ & M[i,j-1] \\ & M[i-1,j-1] + eq(x_i,y_j)\} \end{aligned}$$

  where $eq(x_i,y_j) = 1$ if $x_i = y_j$, $eq(x_i,y_j) = 0$ otherwise.

The following is the content of the matrix for the words `algorithm` and `parachute`.

	λ	p	a	r	a	c	h	u	t	e
λ	0	0	0	0	0	0	0	0	0	0
a	0	0	1	1	1	1	1	1	1	1
l	0	0	1	1	1	1	1	1	1	1
g	0	0	1	1	1	1	1	1	1	1
o	0	0	1	1	1	1	1	1	1	1
r	0	0	1	2	2	2	2	2	2	2
i	0	0	1	2	2	2	2	2	2	2
t	0	0	1	2	2	2	2	2	3	3
h	0	0	1	2	2	2	3	3	3	3
m	0	0	1	2	2	2	3	3	3	3

The matrix can be filled in $O(nm)$ time. How do you reconstruct the longest common substring given the matrix?

3 Chain Matrix Multiplication

Suppose that you want to multiply four matrices $A \times B \times C \times D$ of dimensions 40×20, 20×300, 300×10, and 10×100, respectively. Multiplying an $m \times n$ matrix by an $n \times p$ matrix takes mnp multiplications (a good enough estimate of the running time).

To multiply these matrices as $(((A \times B) \times C) \times D)$ takes $40 \cdot 20 \cdot 300 + 40 \cdot 300 \cdot 10 + 40 \cdot 10 \cdot 100 = 380,000$. A more clever way would be to multiply them as $(A \times ((B \times C) \times D))$, with total cost $20 \cdot 300 \cdot 10 + 20 \cdot 10 \cdot 100 + 40 \cdot 20 \cdot 100 = 160,000$. An even better order would be $((A \times (B \times C)) \times D)$ with total cost $20 \cdot 300 \cdot 10 + 40 \cdot 20 \cdot 10 + 40 \cdot 10 \cdot 100 = 108,000$. Among the five possible orders (the five possible binary trees with four leaves) this latter method is the best.

How can we automatically pick the best among all possible orders for multiplying n given matrices? Exhaustively examining all binary trees is impractical: There are $C(n) = \frac{1}{n}\binom{2n-2}{n-1} \approx \frac{4^n}{n\sqrt{n}}$ such trees ($C(n)$ is called the *Catalan* number of n). Naturally enough, dynamic programming is the answer.

Suppose that the matrices are $A_1 \times A_2 \times \cdots \times A_n$, with dimensions, respectively, $m_0 \times m_1, m_1 \times m_2, \ldots m_{n-1} \times m_n$. Define a *subproblem* (remember, this is the most crucial and nontrivial step in the design of a dynamic programming algorithm; the rest is usually automatic) to be to multiply the matrices $A_i \times \cdots \times A_j$, and let $M(i, j)$ be the optimum number of multiplications for doing so. Naturally, $M(i, i) = 0$, since it takes no effort to multiply a chain consisting just of the i-th matrix. The recursive equation is

$$M(i, j) = \min_{i \le k < j} [M(i, k) + M(k+1, j) + m_{i-1} \cdot m_k \cdot m_j].$$

This equation defines the program and its complexity—$O(n^3)$.

for $i := 1$ to n do $M(i, i) := 0$

for $d := 1$ to $n - 1$ do

for $i := 1$ to $n - d$ do

$j = i + d, M(i, j) = \infty$, best$(i, j)$:=nil
for $k := i$ to $j - 1$ do
  if $M(i, j) > M(i, k) + M(k + 1, j) + m_{i-1} \cdot m_k \cdot m_j$ then
    $M(i, j) := M(i, k) + M(k + 1, j) + m_{i-1} \cdot m_k \cdot m_j$, best$(i, j) := k$

As usual, improvements are possible (in this case, down to $O(n \log n)$).

Run this algorithm in the simple example of four matrices given to verify that the claimed order is the best!