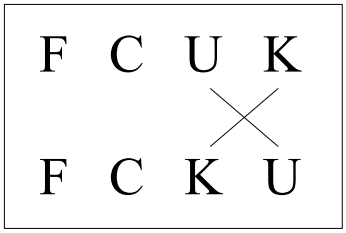
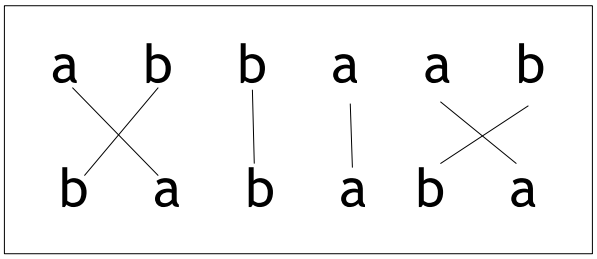
Algorithms Seminar 29th April 2005

Scribed by Sudipto Mukherjee

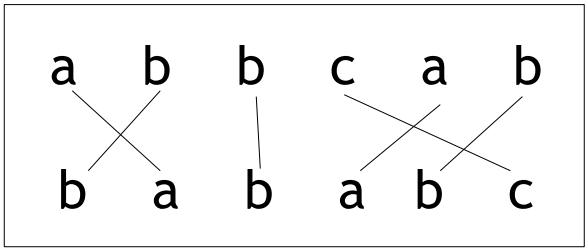
Consider the word FCUK. It is just one transposition away from a more colorful word.
Input: Pattern of length 'n'. The quality of a match is the total distance the letter shave to move.



Distance = 2



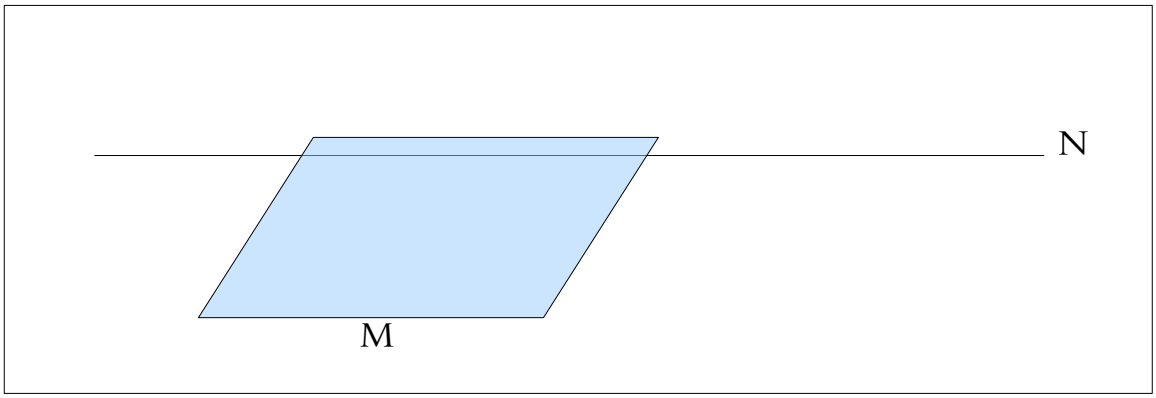
Distance = 4



Distance = 6

It was pointed out that this was the Earth Mover's metric on strings. We need to answer the question, "What is the best match for our pattern in the original string?".

Observation: The distance is always twice the number of inversions.
Let d be the distance between the strings.



Assumption: The alphabet is constant.
Observation: For any given offset, we can calculate the best match.

Naïve method: Perform a bipartite matching for each window.
This is of $O(NM^3)$ complexity.

Once we assume the alphabet to have just 0's and 1's, the problem reduces to the necklace problem we solved earlier in the semester.

Observation: Whether a d=0 match exists or not can be detected in $O(n+m)$ time using any of the popular string matching methods.

For binary strings, we were able to solve the problem in $O(NM)$ time. We should be able to replicate this for this problem, assuming the alphabet is constant.

As we observed for the necklace problem earlier, **a greedy matching will be optimal** in this case. This is because in any given window, matching the leftmost 1 in the window with the leftmost 1 in the search string is always optimal. We cannot get a better result by matching another 1 in the search string.

This assumption should let us reduce the $O(NM^3)$ time to $O(NM)$ time.

Idea: Let us look at windows where the character content is a permutation of the search string.

To do this, we keep an incremental count of how many many characters of each type are there in our current window. Since the alphabet is constant, we can check this in linear time.

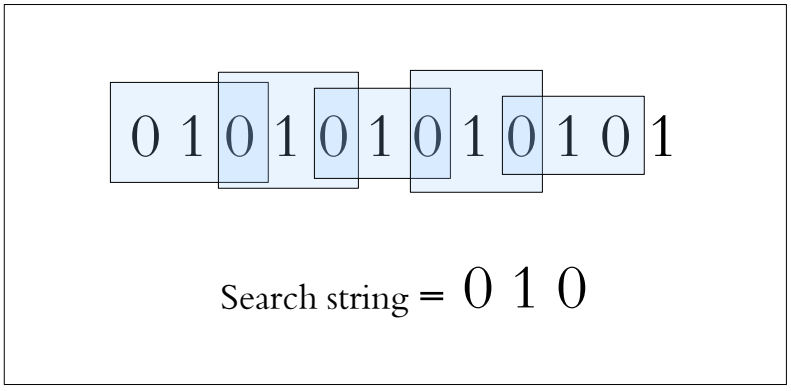
Observation: We have all the windows with a permutation of the search string in linear time. Let there be W such candidate windows obtained in $O(N)$ time.

We need $O(N + WM)$ time to solve the problem.

Idea: Can we improve this using some incremental technique?

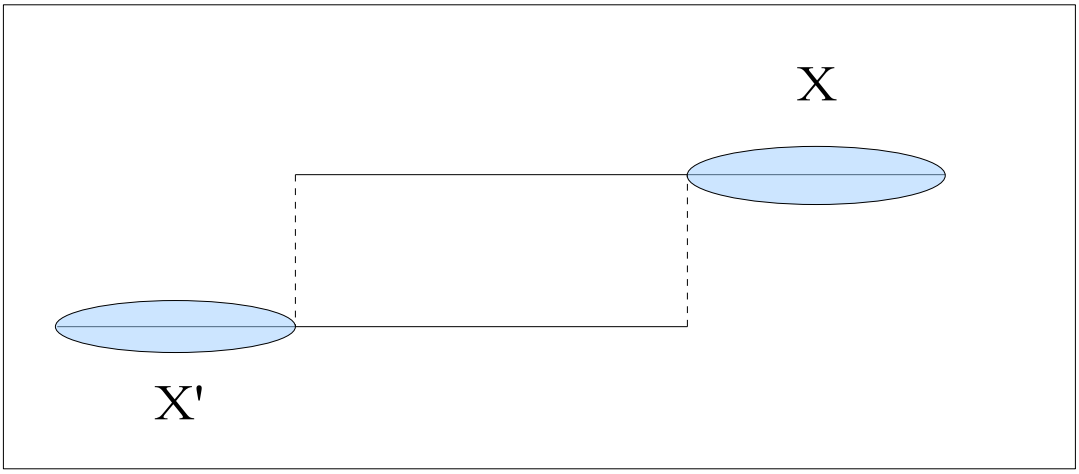
Observation: If the windows did not overlap, W could be less than N/M and we would solve the problem in linear time. The value of W could be as large as $O(N)$ in the worst case. Depending on the value of W, the time taken would vary from $O(N)$ and $O(N^2)$.

Idea: We need to place an exact bound on the worst case value of W.



Collect all groups of windows that overlap and do the distance computations on $O(M)$ time for each such group. Then, we have $O_(N+kM)$ time, where k=depth.

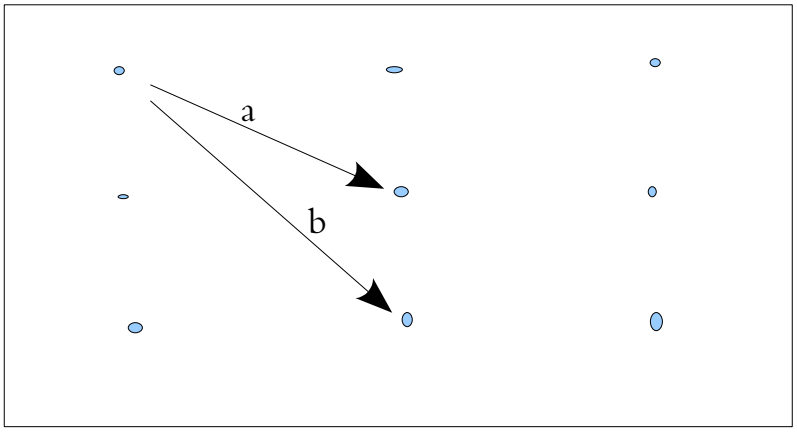
If we have consecutive windows, we have to gain the same character we just lost.
i.e for any overlapping window, we have the following property:



Therefore, X and X' are permutations of each other. We can then possibly use **suffix trees** in some way.

Idea: What if M is very small. It might then be practical to enumerate all the permutations in a table. A matrix of the scores would be maintained, and we use something resembling a finite state machine to walk from one window to another.

This might be useful when $M < (\log 2/M)$
There are about $\sqrt{N}$ possible windows.

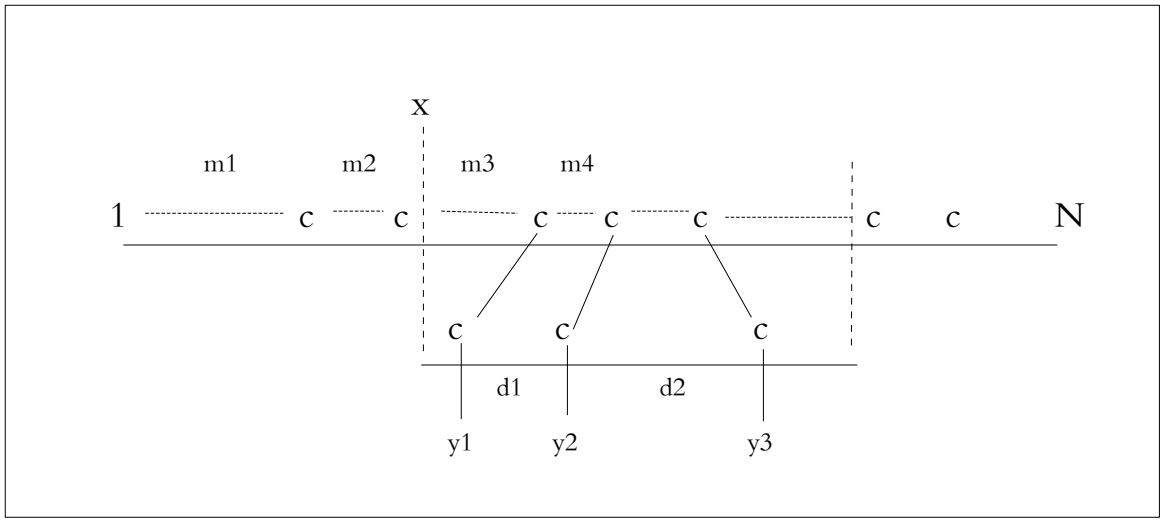


We could walk through this graph and keep track of the lowest weight node we hit as the best match.

Idea: Could be perform the computations for overlapping windows by using suffix trees?

So now, can we do the incremental updating of the cost in constant or sub-linear time?

Event queue idea



We number the characters in the string from 1 to N.
For matching a character 'c', we have:

$$d_1 = |l_1 - y_1|$$
$$d_2 = |l_2 - y_2|$$
$$\ldots\ldots$$
$$d_{kc} = |l_{kc} - y_{kc}|$$

$$\sum_{i=1}^{k_c} d_i = \sum_{i=1}^{k_c} |l_i - y_i|$$

Our cost is now Σd_i at each window using rubber bands.

Events

- Rubber band hits zero i.e. We have a character match
- 'c' falls off to the left
- 'c' falls off to the right

To make the matching constant time, we need two types of rubber bands, ones to the left and ones to the right. For a left rubber band to change to a right rubber band, we must hit the zero state.