# 15-451/651 Algorithms, Fall 2017
# Recitation #5 Worksheet

---

## A String Matching Oracle

In this recitation we generalize the fingerprinting method described in lecture. Given a string $T = t_0, t_1, \ldots, t_N$, let $T_{i,j}$ denote the substring $t_i, t_{i+1}, \ldots, t_j$. We want to preprocess $T$ such that the following test can be answered (with a low probability of a false positive) in constant time:

$$\text{Test if } T_{i,i+\ell} = T_{j,j+\ell}$$

First of all let's define the fingerprinting function. Let $p$ be a (preferably large) prime, along with a base $b$ (larger than the alphabet size). If $S = s_0, s_1, \ldots, s_{n-1}$ is a string over some alphabet of $\Sigma = \{0, 1, \ldots, k-1\}$, then the Karp-Rabin fingerprint of $S$ is:

$$h(S) = (s_0 b^{n-1} + s_1 b^{n-2} + \cdots + s_{n-1} b^0) \bmod p$$

(From now on we will omit the mod$p$ from these expressions.)

Now, to preprocess the string $T$, we will compute the following arrays for $0 \le i \le N$:

$$\begin{aligned} r[i] &= b^i \\ a[i] &= t_0 b^{i-1} + t_1 b^{i-2} + \cdots + t_{i-1} b^0 \end{aligned}$$

Give algorithms for computing these in time $O(N)$:

Now write a simple expression for $h(T_{i,j})$ in terms of the $r[]$ and $a[]$ arrays computed above.

Okay, the answer to the previous part is:

$$h(T_{i,j}) = a[j+1] - a[i] \cdot r[j-i+1]$$

Prove it:

So the end result is that we can test if $T_{i,i+\ell} = T_{j,j+\ell}$ by comparing $h(T_{i,i+\ell})$ with $h(T_{j,j+\ell})$. The probability of a false positive can be made as small as desired by picking a sufficiently large prime $p$, as seen in lecture. (Today we are not concerned with bounding the false positive probability.)

## Extension to string comparison

Suppose we want to know not just if $T_{i,i+\ell}$ equals $T_{j,j+\ell}$, but we want to know the result of comparing these two strings. (That is we want to know if the first is less, equal to, or greater than the second.)

Give an algorithm to do this that runs in $O(\log \ell)$ time:

Finally show how to use what we have developed in these notes to compute the suffix array and the longest common prefix array of $T$ in $O(N \log^2 N)$ time.

## The "Surrogate" Algorithm for Computing Suffix Arrays

If you have time, discuss the algorithm for computing suffix arrays that appears on the next page. Here are some notes about it.

First of all the ocaml code `let w = Array.make n ((0,0),0) in` means to make an array `w` of length `n`. Each element is a pair consisting of a pair of integers, and an integer. Later on this array is sorted. The comparison of a pair is done in the natural lexicographic fashion: The first part of the pairs are compared, and if equal it moves onto the second.

The notation `fst w.(j)` means look up the `j`th element of the array `w` and extract the first part of the pair. In this case it's the pair `(0,0)` assuming the array has not changed since being initialized. `snd w.(j)` returns the second part of the pair.

One subtle part of the code is this:

```
for j=1 to n-1 do
   x.(j) <- x.(j-1) + (if fst w.(j-1) = fst w.(j) then 0 else 1)
done;
```

This constructs the "surrogate" array `x` that replaces the pairs in `w` as far as comparisons are concerned. For example, here is the `w` array (after sorting) and the resulting `x` array.

```
w = ((1,-1),6) ((1,1),4) ((1,1),5) ((2,2),0) ((2,2),1) ((2,3),2) ((3,1),3)

x =      0        1         1         2         2         3         4
```

```
(*
   An Ocaml implementation of the "merging blocks" algorithm
   to compute the suffix array in O(n log^2 n).

   Actually it might better be called the "surrogate" algorithm.
   Because during each pass it replaces a pair (i,j) with a single
   integer, the surrogate.  Comparing the surrogates for two pairs
   gives the same result as comparing the original pairs.  Applying
   this replacement k times means that the surrogate at position i
   represents the string of length 2^k starting at i, for purposes of
   comparison with any other substring of length 2^k.  And it works
   in constant time.

   This algorithm's performance can be improved to O(n log n) by
   replacing the sorting algorithm (in all but the first pass) with a
   2-pass radix sort, since in that case we're sorting pairs (i,j)
   where 0 <= i,j < n.

                    Danny Sleator, October 13, 2016
*)

let suffix_array a_in n =
  (* input is an array of length n of non-negative integers *)
  let small = -1 in  (* less than any number in the input array *)
  let a = Array.copy a_in in
  let w = Array.make n ((0,0),0) in
  let x = Array.make n 0 in  (* surrogates *)
  let rec pass shift =
    let get i = if i+shift < n then a.(i+shift) else small in
    for i=0 to n-1 do
      w.(i) <- ((a.(i), get i), i)
    done;
    Array.fast_sort compare w;
    for j=1 to n-1 do
      x.(j) <- x.(j-1) + (if fst w.(j-1) = fst w.(j) then 0 else 1)
    done;
    if x.(n-1) < n-1 then (
      for j=0 to n-1 do
        a.(snd w.(j)) <- x.(j)
      done;
      pass (shift*2)
    )
  in
  pass 1;
  Array.init n (fun i -> snd w.(i))

(*
   Example:
   Input:  [1, 1, 2, 2, 2, 2, 1, 1, 1, 1]
   Output: [9, 8, 7, 6, 0, 1, 5, 4, 3, 2]
*)
```