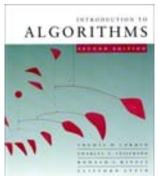
# *Introduction to Algorithms*

**6.046J/18.401J**




## *Lecture 15*

**Prof. Piotr Indyk**

## String Matching

- Input: Two strings T[1…n] and P[1…m], containing symbols from alphabet Σ.
  E.g. :
  - Σ={a,b,…,z}
  - T[1…18]="to be or not to be"
  - P[1..2]="be"
- Goals:
  - Find all "shifts" $0 \leq s \leq n\text{-}m$ such that T[s+1…s+m]=P
  - Find one (e.g., the first) shift



## Plan

- Simple algorithm
  - Worst-case vs. average case
- Karp-Rabin algorithm
  - Randomized "Monte Carlo" algorithm
    - Efficient in the worst case
    - Small probability of error



## Simple Algorithm

```
for s ← 0 to n-m
    Match ← 1
    for j ← 1 to m
        if T[s+j] ≠P[j] then
            Match ← 0
            exit loop
    if Match=1 then output s
```



## Results

- Running time of the simple algorithm:
  - Worst-case: O(nm)
  - Average-case (random text): O(n)
    - $T_s$= time spent on checking shift s
      - Each text character matches pattern character with probability p=1/|Σ|
      - $T_s$ has a geometric distribution
      - $E[T_s] = 1/(1\text{-}p) \leq 2$
    - Expected total time:
      $E\,[\Sigma_s T_s] = \Sigma_s\, E[T_s] = O(n)$


```
for s ← 0 to n-m
    Match ← 1
    for j ← 1 to m
        if T[s+j] ≠P[j] then
            Match ← 0
            exit loop
    if Match=1 then output s
```




## Worst-case

- Is it possible to achieve O(n) for any input ?
  - Knuth-Morris-Pratt'77: deterministic
  - Karp-Rabin'81: randomized Monte Carlo
    - Small probability of error

# Karp-Rabin Algorithm

(c) Piotr Indyk and Manolis Kellis

## Karp-Rabin Algorithm

- A very elegant use of an idea that we have encountered before, namely…

  HASHING !
- Idea:
  - Hash all substrings

    T[1…m], T[2…m+1], …, T[m-n+1…n]
  - Hash the pattern P[1…m]
  - Report the substrings that hash to the same value as P
- Problem: how to hash n-m substrings, each of length m, in O(n) time ?

(c) Piotr Indyk and Manolis Kellis

## Digression

- In previous lectures, we have seen

  $$h_a(x)=\Sigma_i a_i x_i \bmod q$$

  where $a=(a_1,\ldots,a_r)$ , $x=(x_1,\ldots,x_r)$
- To implement it, we would need to compute

  $$h_a(\ T[s\ldots s+m-1]\ )=\Sigma_i a_i\ T[s+i] \bmod q$$

  for $s=0\ldots n-m$
- How to compute it in O(n) time ?
- A big open problem! (see later lecture on FFT)

(c) Piotr Indyk and Manolis Kellis

## Implementation

- Attempt I:
  - Assume $\Sigma=\{0,1\}$
  - Think about each $T_s=T[s+1\ldots s+m]$ as a number in binary representation, i.e.,

    $$t_s=T[s+1]2^{m-1}+T[s+2]2^{m-2}+\ldots+T[s+m]2^0$$
  - Find a fast way of computing $t_{s+1}$ given $t_s$
  - Output all s such that $t_s$ is equal to the number p represented by P

(c) Piotr Indyk and Manolis Kellis

## The great formula

- How to transform

  $$t_s=\underline{T[s+1]2^{m-1}}+T[s+2]2^{m-2}+\ldots+T[s+m]2^0$$

  into

  $$t_{s+1}=T[s+2]2^{m-1}+T[s+3]2^{m-2}+\ldots+\underline{T[s+m+1]2^0}\ ?$$
- Three steps:
  - Subtract $T[s+1]2^{m-1}$
  - Multiply by 2 (i.e., shift the bits by one position)
  - Add $T[s+m+1]2^0$
- Therefore: $t_{s+1}=(t_s-T[s+1]2^{m-1})*2+T[s+m+1]2^0$

(c) Piotr Indyk and Manolis Kellis

## Algorithm

$$t_{s+1}=(t_s-T[s+1]2^{m-1})*2+T[s+m+1]2^0$$

- Can compute $t_{s+1}$ from $t_s$ using 3 arithmetic operations
- Therefore, we can compute all $t_0,t_1,\ldots,t_{n-m}$ using O(n) arithmetic operations
- We can compute a number corresponding to P using O(m) arithmetic operations
- Are we done ?

(c) Piotr Indyk and Manolis Kellis

## Problem

- To get $O(n)$ time, we would need to perform each arithmetic operation in $O(1)$ time
- However, the arguments are $m$-bit long !
- If $m$ large, it is unreasonable to assume that operations on such big numbers can be done in $O(1)$ time
- We need to reduce the number range to something easier to manage

(c) Piotr Indyk and Manolis Kellis

## Attempt II

- We will instead compute
  $t'_s = T[s+1]2^{m-1} + T[s+2]2^{m-2} + \ldots + T[s+m]2^0 \bmod q$
  where $q$ is an "appropriate" prime number
- One can still compute $t'_{s+1}$ from $t'_s$ :
  $t'_{s+1} = (t'_s - T[s+1]2^{m-1})*2 + T[s+m+1]2^0 \bmod q$
- If $q$ is not large, e.g., has $O(\log n)$ bits, we can compute all $t'_s$ (and $p'$) in $O(n)$ time

(c) Piotr Indyk and Manolis Kellis

## Problem

- Unfortunately, we can have false positives, i.e., $T_s \neq P$ but $t_s \bmod q = p \bmod q$
- Need to use a random $q$
- We will show that the probability of a false positive is small

  → randomized Monte Carlo algorithm

(c) Piotr Indyk and Manolis Kellis

## False positives: analysis

- Consider any $t_s \neq p$. We know that both numbers are in the range $\{0 \ldots 2^m - 1\}$
- How many primes $q$ are there such that
  $t_s \bmod q = p \bmod q \;\equiv\; (t_s - p) = 0 \bmod q$ ?
- Such prime has to divide $x = (t_s - p) \leq 2^m$
- Represent $x = p_1^{e1} p_2^{e2} \ldots p_k^{ek}$, $p_i$ prime, $e_i \geq 1$
  What is the largest possible value of $k$ ?
  - Since $2 \leq p_i$, we have $x \geq 2^k$
  - At the same time, $x \leq 2^m$
  - Therefore $k \leq m$
- There are $\leq m$ primes dividing $x$

(c) Piotr Indyk and Manolis Kellis

## Algorithm + Analysis

- Algorithm:
  - Let $\prod$ be a set of $2nm$ primes
  - Choose $q$ uniformly at random from $\prod$
  - Compute $t_0 \bmod q$, $t_1 \bmod q$, …., and $p \bmod q$
  - Report $s$ such that $t_s \bmod q = p \bmod q$
- Analysis:
  - For each $s$, the probability that $T_s \neq P$ but
    $t_s \bmod q = p \bmod q$
    is at most $m/(2nm) = 1/(2n)$
  - From previous slide, the probability of *any* false positive is at most
    $(n-m)/(2n) \leq 1/2$
  - Can replace $2$ by any desired parameter

(c) Piotr Indyk and Manolis Kellis

## "Details"

- Our algorithm uses a prime $q$ chosen uniformly at random from a set $\prod$ of $2nm$ primes
- Two questions:
  - How do we know that such $\prod$ exists ?
    (That is, a set of $2nm$ primes, each having $O(\log n)$ bits)
  - How do we choose a random prime from $\prod$ in $O(n)$ time ?
- We will see only a "sketch" of an answer
  (details require pretty deep theory)
- In practice, just select "large enough" prime in advance

(c) Piotr Indyk and Manolis Kellis

## Optional material



## Prime density

- Primes are "dense". I.e., if PRIMES(N) is the set of primes smaller than N, then asymptotically

  $|PRIMES(N)|/N \sim 1/\ln N$
- If N large enough, then

  $|PRIMES(N)| \geq N/(2\ln N)$

- Proof: Trust me.



## Prime density continued

- Set $N=C\, mn \ln(mn)$
- There exists $C=O(1)$ such that

  $N/(2\ln N) \geq 2mn$
- Proof:

  $C\, mn \ln(mn) / [2 \ln(C\, mn \ln(mn))]$

  $\geq C\, mn \ln(mn) / [2 \ln(C\, (mn)^2)]$

  $= C\, mn \ln(mn) / [4 \ln(C) + 4 \ln(mn)]$

  which is greater than $2mn$ for $C$ large enough
- All elements of PRIMES(N) are $\log N = O(\log n)$ bits long



## Prime selection

- Still need to find a random element of PRIMES(N)
- Solution:
  - Choose a random element from $\{1 \ldots N\}$
  - Check if it is prime
  - If not, repeat
- Analysis:
  - From prime density theorem, a random element $q$ from $\{1\ldots N\}$ is prime with probability $\sim 1/\ln N$
  - We can check if $q$ is prime in time polynomial in $\log N$ :
    - Randomized: Rabin, Solovay-Strassen in 1976
    - Deterministic: Agrawal et al in 2002
- Therefore, we can generate and verify a random prime $q$ in $\log^{O(1)} n$ time



## Final Algorithm

- Set $N=C\, mn \ln(mn)$
- Repeat
  - Choose $q$ uniformly at random from $\{1\ldots N\}$
- Until $q$ is prime
- Compute $t_0 \bmod q$, $t_1 \bmod q$, …., and $p \bmod q$
- Report $s$ such that $t_s \bmod q = p \bmod q$