CSE 566 Spring 2023

String Matching

(Following Gusfield Chapter 2)

(Slides copied/edited from these by Dr. Carl Kingsford)

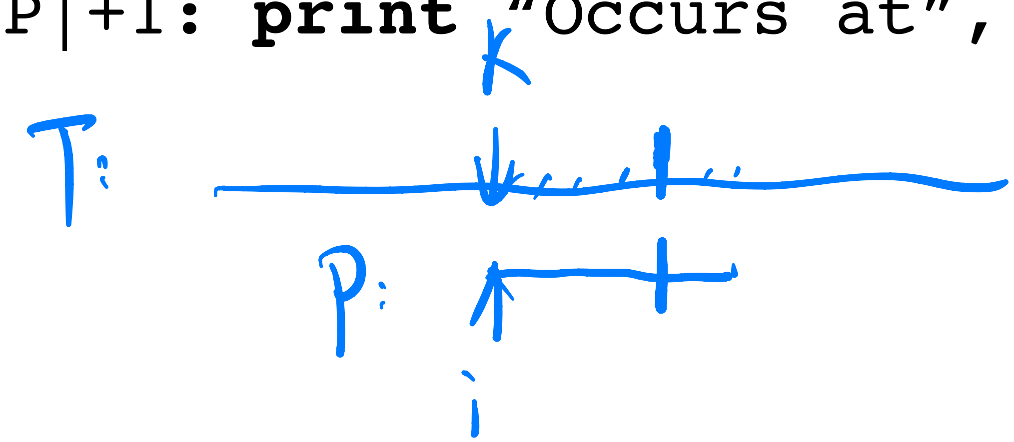
Exact String Matching

Exact String Matching Problem. Given a (long) string T and a shorter string P, find all occurrences of P in T. Occurrences of P are allowed to overlap.

- Example:
  - input: $\boldsymbol{T}$ = *GACTACGACTACTACG*, $\boldsymbol{P}$ = *ACTAC*
  - Output: pattern occurs at positions 2, 8, 11
- A fundamental question:
  - search for words in long documents, webpages, etc.
  - find substrings of DNA, proteins that are known to be important.

The Simple (Slow) Algorithm

```
SimpMatch(T, P):
  for k = 1..|T|:
    i = 1
    while i ≤ |P| and T[k+i-1] == P[i]: i++
    if i==|P|+1: print "Occurs at", k
```



- Runs in $O(|T| \times |P|)$ time.
- Information gathered in **while** loop at iteration i is ignored in iteration i+1.
- Ideas for speeding-up: reuse the information T/P in the **while** loop
  - to avoid unnecessary comparisons in the while-loop (Z algorithm)
  - to increment k by more than 1 in the outer loop, or equivalently, shift P to the right by more than 1 (KMP algorithm)

Exploiting Patterns in P

```
            k
            ↓
T:  All this happened, more or less.
P:           happy
              happy
```

- After comparing "happy" to "happe" at iteration k,
  - we know that $T[k...k+3]$ = "happ" = $P[1...4]$
  - we can deduce that there can be no match at $k+1$ because $T[k+1]$ = P[2] = "a" but $P[1]$ = "h"
  - in fact, since "h" does not appear in $T[k...k+3] = P[1...4]$, we could set $k = k + 4$

- Since T will have matched some part of P, it is the similarities between various parts of P that allow us to make these deductions.

$\Longrightarrow$ Preprocess P to find these similarities.

Z Algorithm

S: ACTTGT

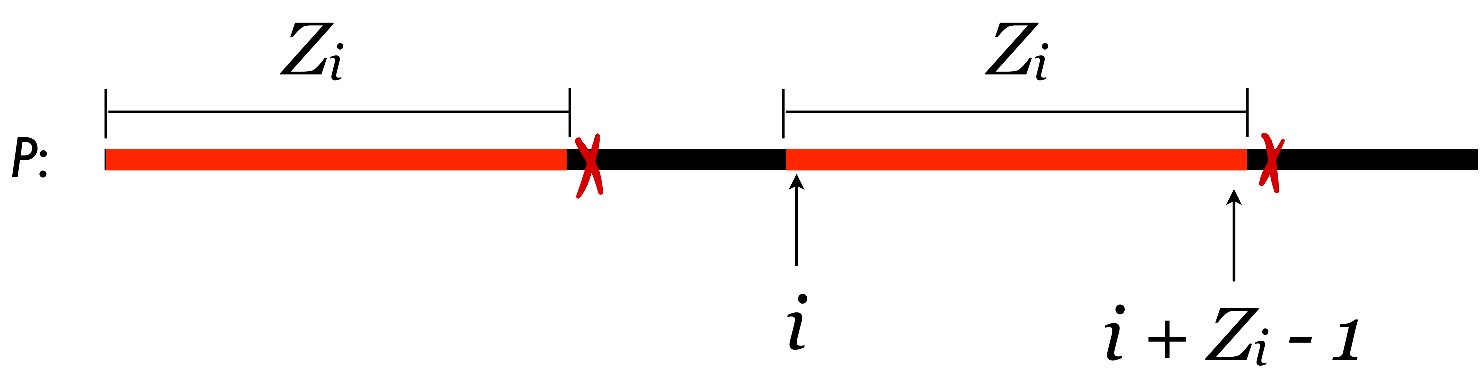
prefix

suffix

Fundamental Preprocessing

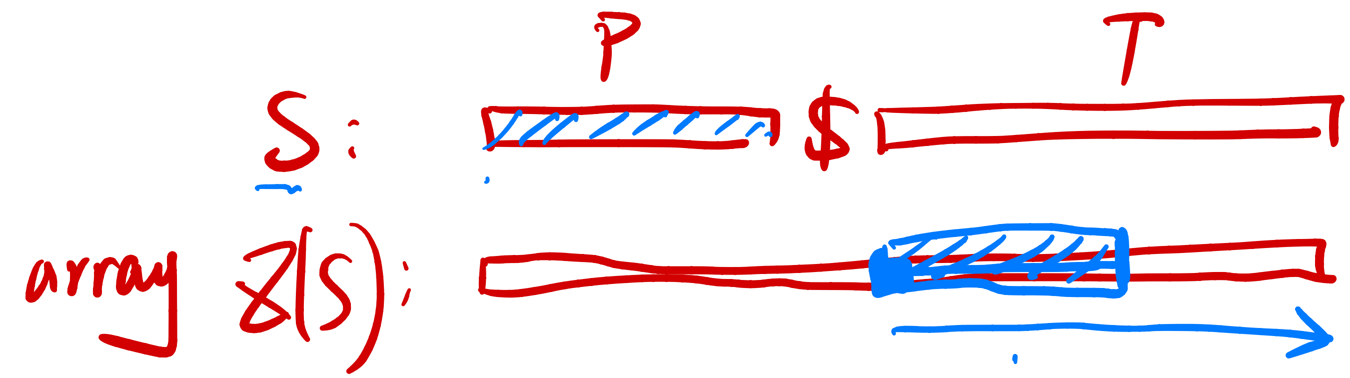
Def. $Z_i(P)$ = the length of the longest substring of P that starts at $i > 1$ and matches a prefix of P.

$Z_1(P)$ is not defined



- P = "aardvark": Z_2 = 1, Z_6 = 1
- P = "alfalfa": Z_4 = 4
- P = "photophosphorescent": Z_6 = 3, Z_{10} = 3

String Search With Z_i




```
ZMatch(T, P):
   S = P$T
   Compute all Z  for S
   for k = 1 to |T|:
      if Z_{k+|P|+1}(S) = |P|: print "Occurs at", k
```

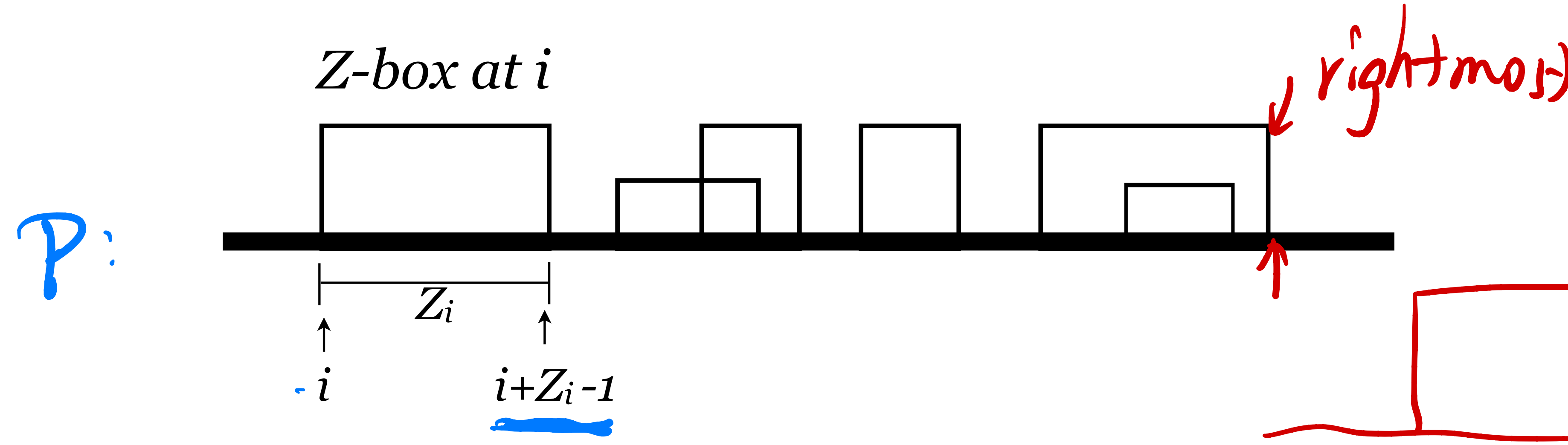
$O(|S|) = O(|P| + |T|)$

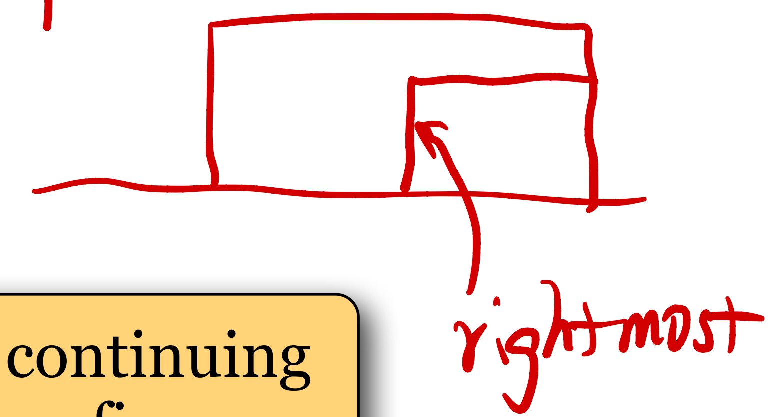
Why does this work?

- $Z_i = |P|$ if and only if the string starting at k matches P. $Z_i(S) \le |P|$
- Running time is O(|P| + |T| + Z_S), where Z_S is the time to compute the Z_i for ~~T~~ S.
- **Next**: an O(|P| + |T|) algorithm for computing the Z_i.

  $= O(|S|)$

Z Boxes





Def. *Z-box at i* is the substring starting at i and continuing to $i+Z_i-1$. This is the substring that matches the prefix. There is no Z-box at i if $Z_i = 0$.

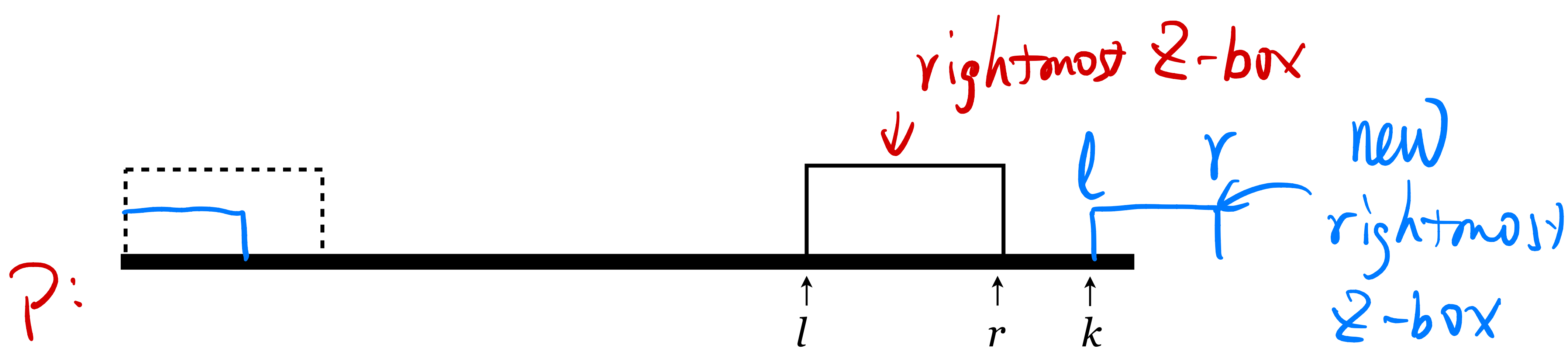
- Algorithm for computing Z_i will iteratively compute Z_k given:
  - $Z_2 \ldots Z_{k-1}$, and
  - the boundaries l, r of the **rightmost** Z-box found starting someplace in $2 \ldots k-1$.

Z Algorithm

- Input: $Z_2 \ldots Z_{k-1}$, and the boundaries l, r of the rightmost Z-box found starting someplace in $2 \ldots k$-1.
- Output: Z_k, and updated l, r of the rightmost Z-box

Case 1: If $k > r$

1. explicitly compute Z_k by comparing with prefix.
2. If $Z_k > 0$: $r = k + Z_k - 1$ and $l = k$ (since this is a new farther right Z-box).

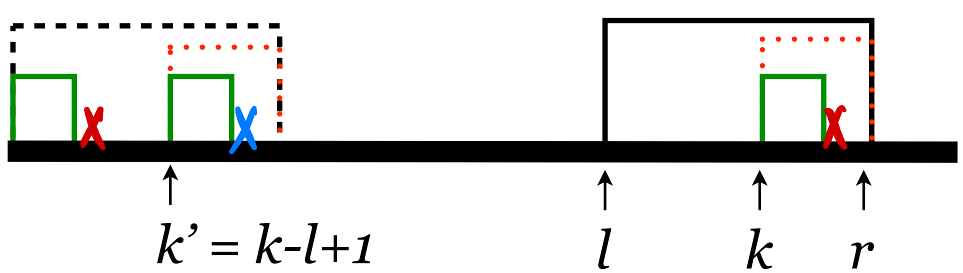


Z Algorithm

Case 2: If $k \leq r$

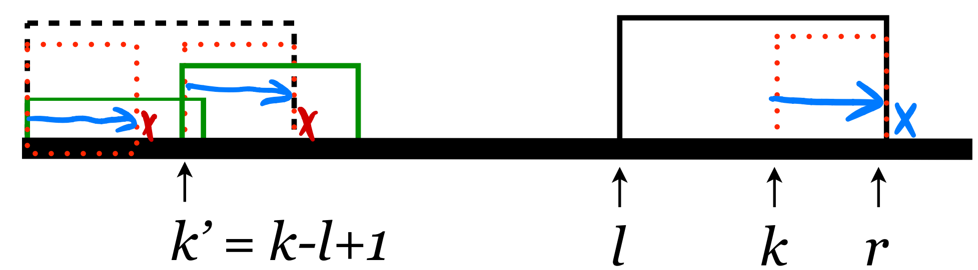
$\beta = r - k + 1$

subcase 1: $Z_{k'} < \beta$:



1. Set $Z_k = Z_{k'}$
2. leave l, r unchanged.

subcase 2: $Z_{k'} \geq \beta$:



1. Explicitly compare after r to set Z_k
2. $l = k$, $r = l + Z_k - 1$

Z Algorithm (pseudo-code)

```
Compute-Z (P):
        l = 0, r = 0, init array Z
        for k = 2 to |P|:
            if k > r:
                let j = 1
                while (P[k + j - 1] = P[j]): j++
                Z[k] = j - 1;
                if Z[k] > 0: l = k and r = l + Z[k] - 1
            else:
                let β = r - k + 1
                let k' = k - l + 1
                if Z[k'] < β:
                     Z[k] = Z[k']
                else:
                     let j = 1
                     while (P[r + j] = P[β + j]): j++
                     Z[k] = β + j - 1
                     l = k
                     r = l + Z[k] - 1
```

✓✓✓✓✓✗

$k \le r$

✓✓✓✓✓✗

(other operations) $\le O(|P|)$

(mismatches) $\le O(|P|)$

(matches) $\le O(|P|)$

Analysis

- Correctness follows by induction and the arguments we made in the description of the algorithm.
- Runs in O(|P|) time:
  - Key observation: characters that are compared in the while-loop are NOT covered by any Z-box.
  - suppose there are j comparisons in a while-loop, then the first j-1 characters are matches and will be covered by the rightmost Z-box.
  - therefore, only match characters covered by a Z-box once, so there are O(|P|) matches in total.
  - every while-loop contains at most one mismatch, so there are O(|P|) mismatches in total.
- Immediately gives an O(|P| + |T|)-time algorithm for string matching as described a few slides ago.
- O(|P| + |T|) is the best possible worst-case running time, since you might have to look at the whole input.