Sublinear Approximate String Matching

Robert Z. West

Department of Informatics
Technische Universität München

Joint Advanced Student School 2004
Sankt Petersburg
Course 1: “Complexity Analysis of String Algorithms”

27th March 2004

This is where you are now.

This is where you are now.

This is where you will end up.

What is that delicacy we want to prepare?

Definition Given a text string T of length n and a pattern string P of length m over a b-letter alphabet, the k-*differences approximate string matching problem* asks for all locations in T where P occurs with at most k differences (substitutions, insertions, deletions).

Example

```
TORTEL LINI
   YELTSIN
   *   **
```

What is that delicacy we want to prepare?

Definition Given a text string T of length n and a pattern string P of length m over a b-letter alphabet, the k-*differences approximate string matching problem* asks for all locations in T where P occurs with at most k differences (substitutions, insertions, deletions).

Example

```
TORTEL LINI
   YELTSIN
   *   **
```

Why are we so hungry?

- Genetics (e.g. GCACTT...) has conjured up new challenges in the field of string processing.
- Sequencing techniques are not perfect: experimental error up to 5–10%.
- Gene mutation (leading to polymorphism) is the mother of evolution. Thus matching a piece of DNA against a database of many individuals must allow a small but significant error.

Why are we so hungry?

- Genetics (e.g. GCACTT...) has conjured up new challenges in the field of string processing.
- Sequencing techniques are not perfect: experimental error up to 5–10%.
- Gene mutation (leading to polymorphism) is the mother of evolution. Thus matching a piece of DNA against a database of many individuals must allow a small but significant error.

Why are we so hungry?

- Genetics (e.g. GCACTT...) has conjured up new challenges in the field of string processing.
- Sequencing techniques are not perfect: experimental error up to 5–10%.
- Gene mutation (leading to polymorphism) is the mother of evolution. Thus matching a piece of DNA against a database of many individuals must allow a small but significant error.

How will we cook the meal?

We will

- first gather the ingredients:
  suffix trees, matching statistics, lowest common ancestor retrieval, edit distance;
- then merge the ingredients and form the algorithm:
  linear expected time algorithm in detail, sublinear expected time after some modifications.

How will we cook the meal?

We will

- first gather the ingredients:
  suffix trees, matching statistics, lowest common ancestor retrieval, edit distance;
- then merge the ingredients and form the algorithm:
  linear expected time algorithm in detail, sublinear expected time after some modifications.

Part I

Gathering the Ingredients

The Auxiliary Tools

Suffix trees

- Remember Olga: She told ya.
- Suffix tree of $P[1..m]\$$: $\mathfrak{S}_P$
- α branching word $\longleftrightarrow$ there are different letters x and y such that both αx and αy are substrings of $P\$$
-

$$\begin{array}{rcl} \text{root} & \longleftrightarrow & \lambda \text{ (empty string)} \\ \{\text{internal nodes}\} & \longleftrightarrow & \{\text{branching words}\} \\ \{\text{leaves}\} & \longleftrightarrow & \{\text{suffixes}\} \end{array}$$

Suffix trees

- Remember Olga: She told ya.
- Suffix tree of $P[1..m]\$$: $\mathfrak{S}_P$
- α branching word $\longleftrightarrow$ there are different letters x and y such that both αx and αy are substrings of $P\$$
-

$$\begin{aligned} \text{root} &\longleftrightarrow \lambda \text{ (empty string)} \\ \{\text{internal nodes}\} &\longleftrightarrow \{\text{branching words}\} \\ \{\text{leaves}\} &\longleftrightarrow \{\text{suffixes}\} \end{aligned}$$

Suffix trees

- Remember Olga: She told ya.
- Suffix tree of $P[1..m]\$$: $\mathfrak{S}_P$
- α branching word $\longleftrightarrow$ there are different letters x and y such that both αx and αy are substrings of $P\$$
-

$$\begin{aligned} \text{root} &\quad\longleftrightarrow\quad \lambda \text{ (empty string)} \\ \{\text{internal nodes}\} &\quad\longleftrightarrow\quad \{\text{branching words}\} \\ \{\text{leaves}\} &\quad\longleftrightarrow\quad \{\text{suffixes}\} \end{aligned}$$

Suffix trees

- Remember Olga: She told ya.
- Suffix tree of $P[1..m]\$$: $\mathfrak{S}_P$
- α branching word $\longleftrightarrow$ there are different letters x and y such that both αx and αy are substrings of $P\$$
-

$$\begin{array}{rcl}
\text{root} & \longleftrightarrow & \lambda \text{ (empty string)} \\
\{\text{internal nodes}\} & \longleftrightarrow & \{\text{branching words}\} \\
\{\text{leaves}\} & \longleftrightarrow & \{\text{suffixes}\}
\end{array}$$

- $\mathrm{floor}(\alpha) :=$ "longest prefix of α that is a branching word"
- $\mathrm{ceil}(\alpha) :=$
  "shortest extension of α that is a branching word or a suffix"
- Note: α branching word $\longleftrightarrow$ $\mathrm{floor}(\alpha) = \mathrm{ceil}(\alpha) = \alpha$
- $\beta^{-1}\alpha :=$ "α without its prefix β"
- Label on edge (β, α): (x, l, r) such that
  $P\$[l] = x$; $\beta^{-1}\alpha = P\$[l..r]$
- $\mathrm{son}(\beta, x) := \alpha$
- $\mathrm{first}(\beta, x) := l$
- $\mathrm{len}(\beta, x) := r - l + 1$
- $\mathrm{shift}(\alpha) :=$ "α without its first letter", if $\alpha \neq \lambda$ (cf. suffix links)

- $\mathrm{floor}(\alpha) :=$ "longest prefix of α that is a branching word"
- $\mathrm{ceil}(\alpha) :=$
  "shortest extension of α that is a branching word or a suffix"
- Note: α branching word $\longleftrightarrow \mathrm{floor}(\alpha) = \mathrm{ceil}(\alpha) = \alpha$
- $\beta^{-1}\alpha :=$ "α without its prefix β"
- Label on edge (β, α): (x, l, r) such that
  $P\$[l] = x; \ \beta^{-1}\alpha = P\$[l..r]$
- $\mathrm{son}(\beta, x) := \alpha$
- $\mathrm{first}(\beta, x) := l$
- $\mathrm{len}(\beta, x) := r - l + 1$
- $\mathrm{shift}(\alpha) :=$ "α without its first letter", if $\alpha \neq \lambda$ (cf. suffix links)

- $\operatorname{floor}(\alpha) :=$ "longest prefix of α that is a branching word"
- $\operatorname{ceil}(\alpha) :=$
  "shortest extension of α that is a branching word or a suffix"
- Note: α branching word $\longleftrightarrow \operatorname{floor}(\alpha) = \operatorname{ceil}(\alpha) = \alpha$
- $\beta^{-1}\alpha :=$ "α without its prefix β"
- Label on edge (β, α): (x, l, r) such that
  $P\$[l] = x$; $\beta^{-1}\alpha = P\$[l..r]$
- $\operatorname{son}(\beta, x) := \alpha$
- $\operatorname{first}(\beta, x) := l$
- $\operatorname{len}(\beta, x) := r - l + 1$
- $\operatorname{shift}(\alpha) :=$ "α without its first letter", if $\alpha \neq \lambda$ (cf. suffix links)

- $\mathrm{floor}(\alpha) :=$ "longest prefix of α that is a branching word"
- $\mathrm{ceil}(\alpha) :=$
  "shortest extension of α that is a branching word or a suffix"
- Note: α branching word $\longleftrightarrow \mathrm{floor}(\alpha) = \mathrm{ceil}(\alpha) = \alpha$
- $\beta^{-1}\alpha :=$ "α without its prefix β"
- Label on edge (β, α): (x, l, r) such that
  $P\$[l] = x;\ \beta^{-1}\alpha = P\$[l..r]$
- $\mathrm{son}(\beta, x) := \alpha$
- $\mathrm{first}(\beta, x) := l$
- $\mathrm{len}(\beta, x) := r - l + 1$
- $\mathrm{shift}(\alpha) :=$ "α without its first letter", if $\alpha \neq \lambda$ (cf. suffix links)

- $\mathrm{floor}(\alpha) :=$ "longest prefix of α that is a branching word"
- $\mathrm{ceil}(\alpha) :=$
  "shortest extension of α that is a branching word or a suffix"
- Note: α branching word $\longleftrightarrow$ $\mathrm{floor}(\alpha) = \mathrm{ceil}(\alpha) = \alpha$
- $\beta^{-1}\alpha :=$ "α without its prefix β"
- Label on edge (β, α): (x, l, r) such that
  $P\$[l] = x;\ \beta^{-1}\alpha = P\$[l..r]$
- $\mathrm{son}(\beta, x) := \alpha$
- $\mathrm{first}(\beta, x) := l$
- $\mathrm{len}(\beta, x) := r - l + 1$
- $\mathrm{shift}(\alpha) :=$ "α without its first letter", if $\alpha \neq \lambda$ (cf. suffix links)

- $\mathrm{floor}(\alpha) :=$ "longest prefix of α that is a branching word"
- $\mathrm{ceil}(\alpha) :=$
  "shortest extension of α that is a branching word or a suffix"
- Note: α branching word $\longleftrightarrow$ $\mathrm{floor}(\alpha) = \mathrm{ceil}(\alpha) = \alpha$
- $\beta^{-1}\alpha :=$ "α without its prefix β"
- Label on edge (β, α): (x, l, r) such that
  $P\$[l] = x;\ \beta^{-1}\alpha = P\$[l..r]$
- $\mathrm{son}(\beta, x) := \alpha$
- $\mathrm{first}(\beta, x) := l$
- $\mathrm{len}(\beta, x) := r - l + 1$
- $\mathrm{shift}(\alpha) :=$ "α without its first letter", if $\alpha \neq \lambda$ (cf. suffix links)

- $\mathrm{floor}(\alpha) :=$ “longest prefix of α that is a branching word”
- $\mathrm{ceil}(\alpha) :=$
  “shortest extension of α that is a branching word or a suffix”
- Note: α branching word $\longleftrightarrow$ $\mathrm{floor}(\alpha) = \mathrm{ceil}(\alpha) = \alpha$
- $\beta^{-1}\alpha :=$ “α without its prefix β”
- Label on edge (β, α): (x, l, r) such that
  $P\$[l] = x;\ \beta^{-1}\alpha = P\$[l..r]$
- $\mathrm{son}(\beta, x) := \alpha$
- $\mathrm{first}(\beta, x) := l$
- $\mathrm{len}(\beta, x) := r - l + 1$
- $\mathrm{shift}(\alpha) :=$ “α without its first letter”, if $\alpha \neq \lambda$ (cf. suffix links)

- $\mathrm{floor}(\alpha) :=$ "longest prefix of α that is a branching word"
- $\mathrm{ceil}(\alpha) :=$
  "shortest extension of α that is a branching word or a suffix"
- Note: α branching word $\longleftrightarrow$ $\mathrm{floor}(\alpha) = \mathrm{ceil}(\alpha) = \alpha$
- $\beta^{-1}\alpha :=$ "α without its prefix β"
- Label on edge (β, α): (x, l, r) such that
  $P\$[l] = x;\ \beta^{-1}\alpha = P\$[l..r]$
- $\mathrm{son}(\beta, x) := \alpha$
- $\mathrm{first}(\beta, x) := l$
- $\mathrm{len}(\beta, x) := r - l + 1$
- $\mathrm{shift}(\alpha) :=$ "α without its first letter", if $\alpha \neq \lambda$ (cf. suffix links)

- $\operatorname{floor}(\alpha) :=$ "longest prefix of α that is a branching word"
- $\operatorname{ceil}(\alpha) :=$
  "shortest extension of α that is a branching word or a suffix"
- Note: α branching word $\longleftrightarrow \operatorname{floor}(\alpha) = \operatorname{ceil}(\alpha) = \alpha$
- $\beta^{-1}\alpha :=$ "α without its prefix β"
- Label on edge (β, α): (x, l, r) such that
  $P\$[l] = x;\ \beta^{-1}\alpha = P\$[l..r]$
- $\operatorname{son}(\beta, x) := \alpha$
- $\operatorname{first}(\beta, x) := l$
- $\operatorname{len}(\beta, x) := r - l + 1$
- $\operatorname{shift}(\alpha) :=$ "α without its first letter", if $\alpha \neq \lambda$ (cf. suffix links)

Matching statistics

Definition The *matching statistics* of text $T[1..n]$ with respect to pattern $P[1..m]$ is an integer vector $\mathfrak{M}_{T,P}$ together with a vector $\mathfrak{M}'_{T,P}$ of pointers to the nodes of $\mathfrak{S}_P$, where $\mathfrak{M}_{T,P}[i] = l$ if l is the length of the longest substring of $P\$$ (anywhere in $P\$$) matching exactly a prefix of $T[i..n]$ and where $\mathfrak{M}'_{T,P}[i]$ points to $\text{ceil}(T[i..i+l-1])$.
More shortly we will write $\mathfrak{M}$ and $\mathfrak{M}'$.

How do we compute the matching statistics?

- Goal: $\mathcal{O}(n+m)$ time algorithm for computing the matching statistics of T and P in a single left-to-right scan of T using just $\mathfrak{S}_P$
- Brief description: The longest match starting at position 1 in T is found by walking down the tree, matching one letter a time.
  Subsequent longest matches are found by following suffix links and carefully going down the tree. (cf. Ukkonen's construction of the suffix tree: "skip-and-count trick")

How do we compute the matching statistics?

- Goal: $\mathcal{O}(n+m)$ time algorithm for computing the matching statistics of T and P in a single left-to-right scan of T using just $\mathfrak{S}_P$
- Brief description: The longest match starting at position 1 in T is found by walking down the tree, matching one letter a time.

  Subsequent longest matches are found by following suffix links and carefully going down the tree. (cf. Ukkonen's construction of the suffix tree: "skip-and-count trick")

How do we compute the matching statistics?

- Goal: $\mathcal{O}(n+m)$ time algorithm for computing the matching statistics of T and P in a single left-to-right scan of T using just $\mathfrak{S}_P$
- Brief description: The longest match starting at position 1 in T is found by walking down the tree, matching one letter a time.
  Subsequent longest matches are found by following suffix links and carefully going down the tree. (cf. Ukkonen's construction of the suffix tree: "skip-and-count trick")

- i, j, k are indices into T:
  - The i-th iteration computes $\mathfrak{M}[i]$ and $\mathfrak{M}'[i]$.
  - Position k of T has just been scanned.
  - j is some position between i and k.
- Invariants:
  - At all times true:
    (1) $T[i..k-1]$ is a substring of P; $T[i..j-1]$ is a branching word of P.
  - After step 3.1 the following becomes true:
    (2) $T[i..j-1] = \text{floor}(T[i..k-1])$ and corresponds to node α.
  - After step 3.2 the following becomes true as well:
    (3) $T[i..k]$ is not a word.

- i, j, k are indices into T:
    - The i-th iteration computes $\mathfrak{M}[i]$ and $\mathfrak{M}'[i]$.
    - Position k of T has just been scanned.
    - j is some position between i and k.
- Invariants:
    - At all times true:
      (1) $T[i..k-1]$ is a substring of P; $T[i..j-1]$ is a branching word of P.
    - After step 3.1 the following becomes true:
      (2) $T[i..j-1] = \text{floor}(T[i..k-1])$ and corresponds to node α.
    - After step 3.2 the following becomes true as well:
      (3) $T[i..k]$ is not a word.

- i, j, k are indices into T:
  - The i-th iteration computes $\mathfrak{M}[i]$ and $\mathfrak{M}'[i]$.
  - Position k of T has just been scanned.
  - j is some position between i and k.
- Invariants:
  - At all times true:
    (1) $T[i..k-1]$ is a substring of P; $T[i..j-1]$ is a branching word of P.
  - After step 3.1 the following becomes true:
    (2) $T[i..j-1] = \text{floor}(T[i..k-1])$ and corresponds to node α.
  - After step 3.2 the following becomes true as well:
    (3) $T[i..k]$ is not a word.

- i, j, k are indices into T:
  - The i-th iteration computes $\mathfrak{M}[i]$ and $\mathfrak{M}'[i]$.
  - Position k of T has just been scanned.
  - j is some position between i and k.
- Invariants:
  - At all times true:
    (1) $T[i..k-1]$ is a substring of P; $T[i..j-1]$ is a branching word of P.
  - After step 3.1 the following becomes true:
    (2) $T[i..j-1] = \text{floor}(T[i..k-1])$ and corresponds to node α.
  - After step 3.2 the following becomes true as well:
    (3) $T[i..k]$ is not a word.

- If $j < k$ after step 3.1, then $T[i..k-1]$ is not a branching word (2), so neither is $T[i-1..k-1]$.
  So, as substrings of P they must have the same single-letter extension.
  We know from iteration $i-1$ that $T[i-1..k-1]$ is a substring of P (1) but $T[i-1..k]$ is not (3), so $T[k]$ cannot be this letter. Hence the match cannot be extended.
- Together invariants (1) and (3) imply $\mathfrak{M}[i] = k - i$.
- i, j, k never decrease and are bounded by n: $i + j + k \leq 3n$.
  For every constant amount of work in step 3, at least one of i, j, k is increased. The running time is therefore $\mathcal{O}(n)$ for step 3, and of course $\mathcal{O}(m)$ for steps 1 and 2, yielding together the desired $\mathcal{O}(n+m)$.

- If $j < k$ after step 3.1, then $T[i..k-1]$ is not a branching word (2), so neither is $T[i-1..k-1]$.
  So, as substrings of P they must have the same single-letter extension.
  We know from iteration $i-1$ that $T[i-1..k-1]$ is a substring of P (1) but $T[i-1..k]$ is not (3), so $T[k]$ cannot be this letter. Hence the match cannot be extended.
- Together invariants (1) and (3) imply $\mathfrak{M}[i] = k - i$.
- i, j, k never decrease and are bounded by n: $i + j + k \leq 3n$.
  For every constant amount of work in step 3, at least one of i, j, k is increased. The running time is therefore $\mathcal{O}(n)$ for step 3, and of course $\mathcal{O}(m)$ for steps 1 and 2, yielding together the desired $\mathcal{O}(n+m)$.

- If $j < k$ after step 3.1, then $T[i..k-1]$ is not a branching word (2), so neither is $T[i-1..k-1]$.
  So, as substrings of P they must have the same single-letter extension.
  We know from iteration $i-1$ that $T[i-1..k-1]$ is a substring of P (1) but $T[i-1..k]$ is not (3), so $T[k]$ cannot be this letter. Hence the match cannot be extended.
- Together invariants (1) and (3) imply $\mathfrak{M}[i] = k - i$.
- i, j, k never decrease and are bounded by n: $i + j + k \leq 3n$.
  For every constant amount of work in step 3, at least one of i, j, k is increased. The running time is therefore $\mathcal{O}(n)$ for step 3, and of course $\mathcal{O}(m)$ for steps 1 and 2, yielding together the desired $\mathcal{O}(n+m)$.

1 construct $\mathfrak{S}_P$ in $\mathcal{O}(m)$ time
2 $\alpha :=$ root; $j := k := 1$
3 **for** $i := 1$ to n **do**
3.1 **while** $(j < k) \wedge (j + \mathrm{len}(\alpha, T[j]) \leq k)$ **do** // *"skip and count"*
$\alpha := \mathrm{son}(\alpha, T[j])$;
$j := j + \mathrm{len}(\alpha, T[j])$
elihw
3.2 **if** $j = k$ **then** // *extend the match*
while $\mathrm{son}(\alpha, T[j])$ exists $\wedge\ T[k] = P\$[\mathrm{first}(\alpha, T[j]) + k - j]$ **do**
$k := k + 1$
if $k = j + \mathrm{len}(\alpha, T[j])$ **then**
$\alpha := \mathrm{son}(\alpha, T[j])$;
$j := k$ **fi**
elihw
fi

3.3 $\mathfrak{M}[i] := k - i$
if $j = k$ **then** $\mathfrak{M}'[i] := \alpha$
  else $\mathfrak{M}'[i] := \mathrm{son}(\alpha, T[j])$ **fi**
3.4 **if** (α is root) $\wedge$ $(j = k)$ **then**
  $j := j + 1$;
  $k := k + 1$ **fi**
if (α is root) $\wedge$ $(j < k)$ **then**
  $j := j + 1$ **fi**
if (α is not root) **then**
  $\alpha := \mathrm{shift}(\alpha)$ **fi**
rof

Lowest common ancestor (LCA) retrieval

Definition For nodes u, v of a rooted tree $\mathfrak{T}$, $\mathrm{LCA}(u, v)$ is the node furthest from the root that is an ancestor to both u and v.

- Goal: constant time LCA retrieval after some preprocessing
- Solution: Reduce the LCA problem to the *range minimum query (RMQ)* problem.

Definition For an array $\mathfrak{A}$ and indices i and j, $\mathrm{RMQ}_{\mathfrak{A}}(i, j)$ is the index of the smallest element in the subarray $\mathfrak{A}[i..j]$.

Lowest common ancestor (LCA) retrieval

Definition For nodes u, v of a rooted tree $\mathfrak{T}$, $\text{LCA}(u, v)$ is the node furthest from the root that is an ancestor to both u and v.

- Goal: constant time LCA retrieval after some preprocessing
- Solution: Reduce the LCA problem to the *range minimum query (RMQ)* problem.

Definition For an array $\mathfrak{A}$ and indices i and j, $\text{RMQ}_{\mathfrak{A}}(i, j)$ is the index of the smallest element in the subarray $\mathfrak{A}[i..j]$.

Lowest common ancestor (LCA) retrieval

Definition For nodes u, v of a rooted tree $\mathfrak{T}$, $\text{LCA}(u, v)$ is the node furthest from the root that is an ancestor to both u and v.

- Goal: constant time LCA retrieval after some preprocessing
- Solution: Reduce the LCA problem to the *range minimum query (RMQ)* problem.

Definition For an array $\mathfrak{A}$ and indices i and j, $\text{RMQ}_{\mathfrak{A}}(i, j)$ is the index of the smallest element in the subarray $\mathfrak{A}[i..j]$.

Lowest common ancestor (LCA) retrieval

Definition For nodes u, v of a rooted tree $\mathfrak{T}$, $\mathrm{LCA}(u, v)$ is the node furthest from the root that is an ancestor to both u and v.

- Goal: constant time LCA retrieval after some preprocessing
- Solution: Reduce the LCA problem to the *range minimum query (RMQ)* problem.

Definition For an array $\mathfrak{A}$ and indices i and j, $\mathrm{RMQ}_{\mathfrak{A}}(i, j)$ is the index of the smallest element in the subarray $\mathfrak{A}[i..j]$.

If an algorithm has preprocessing time $p(n)$ and query time $q(n)$, we say it has complexity $\langle p(n), q(n)\rangle$.

Lemma If there is a $\langle p(n), q(n)\rangle$-time solution for RMQ on a length n array, then there is a $\langle \mathcal{O}(n) + p(2n-1), \mathcal{O}(1) + q(2n-1)\rangle$-time solution for LCA in a tree with n nodes.

The $\mathcal{O}(n)$ term will come from the time needed to create the soon-to-be-presented arrays.
The $\mathcal{O}(1)$ term will come from the time needed to convert the RMQ answer on one of these arrays to the LCA answer in the tree.

If an algorithm has preprocessing time $p(n)$ and query time $q(n)$, we say it has complexity $\langle p(n), q(n) \rangle$.

Lemma If there is a $\langle p(n), q(n) \rangle$-time solution for RMQ on a length n array, then there is a $\langle \mathcal{O}(n) + p(2n-1), \mathcal{O}(1) + q(2n-1) \rangle$-time solution for LCA in a tree with n nodes.

The $\mathcal{O}(n)$ term will come from the time needed to create the soon-to-be-presented arrays.
The $\mathcal{O}(1)$ term will come from the time needed to convert the RMQ answer on one of these arrays to the LCA answer in the tree.

If an algorithm has preprocessing time $p(n)$ and query time $q(n)$, we say it has complexity $\langle p(n), q(n) \rangle$.

Lemma If there is a $\langle p(n), q(n) \rangle$-time solution for RMQ on a length n array, then there is a $\langle \mathcal{O}(n) + p(2n-1), \mathcal{O}(1) + q(2n-1) \rangle$-time solution for LCA in a tree with n nodes.

The $\mathcal{O}(n)$ term will come from the time needed to create the soon-to-be-presented arrays.
The $\mathcal{O}(1)$ term will come from the time needed to convert the RMQ answer on one of these arrays to the LCA answer in the tree.

Proof The LCA of nodes u and v is the shallowest (i.e. closest to the root) node between the visits to u and v encountered during a depth first search (DFS) traversal of $\mathfrak{T}$ (n nodes; labels: $1, \ldots, n$).
Therefore, the reduction proceeds as follows:

1. Let array $\mathfrak{D}[1..2n-1]$ store the nodes visited in a DFS of $\mathfrak{T}$. $\mathfrak{D}[i]$ is the label on the i-th node visited in the DFS.
2. Let the *level* of a node be its distance from the root. Compute the level array $\mathfrak{L}[1..2n-1]$, where $\mathfrak{L}[i]$ is the level of node $\mathfrak{D}[i]$.
3. Let the *representative* of a node be the index of its first occurrence in the DFS. Compute the representative array $\mathfrak{R}[1..n]$, where $\mathfrak{R}[w] = \min\{j \mid \mathfrak{D}[j] = w\}$.

Feasible during a single DFS; thus running time $\mathcal{O}(n)$.

Proof The LCA of nodes u and v is the shallowest (i.e. closest to the root) node between the visits to u and v encountered during a depth first search (DFS) traversal of $\mathfrak{T}$ (n nodes; labels: $1, \ldots, n$). Therefore, the reduction proceeds as follows:

1. Let array $\mathfrak{D}[1..2n-1]$ store the nodes visited in a DFS of $\mathfrak{T}$. $\mathfrak{D}[i]$ is the label on the i-th node visited in the DFS.
2. Let the *level* of a node be its distance from the root. Compute the level array $\mathfrak{L}[1..2n-1]$, where $\mathfrak{L}[i]$ is the level of node $\mathfrak{D}[i]$.
3. Let the *representative* of a node be the index of its first occurrence in the DFS. Compute the representative array $\mathfrak{R}[1..n]$, where $\mathfrak{R}[w] = \min\{j \mid \mathfrak{D}[j] = w\}$.

Feasible during a single DFS; thus running time $\mathcal{O}(n)$.

Proof The LCA of nodes u and v is the shallowest (i.e. closest to the root) node between the visits to u and v encountered during a depth first search (DFS) traversal of $\mathfrak{T}$ (n nodes; labels: $1, ..., n$). Therefore, the reduction proceeds as follows:

1. Let array $\mathfrak{D}[1..2n-1]$ store the nodes visited in a DFS of $\mathfrak{T}$. $\mathfrak{D}[i]$ is the label on the i-th node visited in the DFS.
2. Let the *level* of a node be its distance from the root. Compute the level array $\mathfrak{L}[1..2n-1]$, where $\mathfrak{L}[i]$ is the level of node $\mathfrak{D}[i]$.
3. Let the *representative* of a node be the index of its first occurrence in the DFS. Compute the representative array $\mathfrak{R}[1..n]$, where $\mathfrak{R}[w] = \min\{j \mid \mathfrak{D}[j] = w\}$.

Feasible during a single DFS; thus running time $\mathcal{O}(n)$.

Proof The LCA of nodes u and v is the shallowest (i.e. closest to the root) node between the visits to u and v encountered during a depth first search (DFS) traversal of $\mathfrak{T}$ (n nodes; labels: $1, ..., n$). Therefore, the reduction proceeds as follows:

1. Let array $\mathfrak{D}[1..2n-1]$ store the nodes visited in a DFS of $\mathfrak{T}$. $\mathfrak{D}[i]$ is the label on the i-th node visited in the DFS.
2. Let the *level* of a node be its distance from the root. Compute the level array $\mathfrak{L}[1..2n-1]$, where $\mathfrak{L}[i]$ is the level of node $\mathfrak{D}[i]$.
3. Let the *representative* of a node be the index of its first occurrence in the DFS. Compute the representative array $\mathfrak{R}[1..n]$, where $\mathfrak{R}[w] = \min\{j \mid \mathfrak{D}[j] = w\}$.

Feasible during a single DFS; thus running time $\mathcal{O}(n)$.

Proof The LCA of nodes u and v is the shallowest (i.e. closest to the root) node between the visits to u and v encountered during a depth first search (DFS) traversal of $\mathfrak{T}$ (n nodes; labels: $1, \ldots, n$). Therefore, the reduction proceeds as follows:

1. Let array $\mathfrak{D}[1..2n-1]$ store the nodes visited in a DFS of $\mathfrak{T}$. $\mathfrak{D}[i]$ is the label on the i-th node visited in the DFS.
2. Let the *level* of a node be its distance from the root. Compute the level array $\mathfrak{L}[1..2n-1]$, where $\mathfrak{L}[i]$ is the level of node $\mathfrak{D}[i]$.
3. Let the *representative* of a node be the index of its first occurrence in the DFS. Compute the representative array $\mathfrak{R}[1..n]$, where $\mathfrak{R}[w] = \min\{j \mid \mathfrak{D}[j] = w\}$.

Feasible during a single DFS; thus running time $\mathcal{O}(n)$.

LCA computed as follows (suppose u is visited before v):

- Nodes between the first visits to u and v: $\mathfrak{D}[\mathfrak{R}[u]..\mathfrak{R}[v]]$
- Shallowest node in this subtour at index $\textsc{rmq}_{\mathfrak{L}}(\mathfrak{R}[u], \mathfrak{R}[v])$
- Node at this position and thus output of $\textsc{lca}(u, v)$: $\mathfrak{D}[\textsc{rmq}_{\mathfrak{L}}(\mathfrak{R}[u], \mathfrak{R}[v])]$

Time complexity as claimed in the lemma:

- Just $\mathfrak{L}$ (size $2n - 1$) must be proprocessed for RMQ. Total preprocessing: $\mathcal{O}(n) + p(2n - 1)$
- For the query: one RMQ in $\mathfrak{L}$ and three constant time array lookups. In total: $\mathcal{O}(1) + q(2n - 1)$. □

LCA computed as follows (suppose u is visited before v):

- Nodes between the first visits to u and v: $\mathfrak{D}[\mathfrak{R}[u]..\mathfrak{R}[v]]$
- Shallowest node in this subtour at index $\mathrm{RMQ}_{\mathfrak{L}}(\mathfrak{R}[u], \mathfrak{R}[v])$
- Node at this position and thus output of $\mathrm{LCA}(u, v)$: $\mathfrak{D}[\mathrm{RMQ}_{\mathfrak{L}}(\mathfrak{R}[u], \mathfrak{R}[v])]$

Time complexity as claimed in the lemma:

- Just $\mathfrak{L}$ (size $2n-1$) must be proprocessed for RMQ. Total preprocessing: $\mathcal{O}(n) + p(2n-1)$
- For the query: one RMQ in $\mathfrak{L}$ and three constant time array lookups. In total: $\mathcal{O}(1) + q(2n-1)$. □

LCA computed as follows (suppose u is visited before v):

- Nodes between the first visits to u and v: $\mathfrak{D}[\mathfrak{R}[u]..\mathfrak{R}[v]]$
- Shallowest node in this subtour at index $\text{RMQ}_{\mathfrak{L}}(\mathfrak{R}[u], \mathfrak{R}[v])$
- Node at this position and thus output of $\text{LCA}(u, v)$: $\mathfrak{D}[\text{RMQ}_{\mathfrak{L}}(\mathfrak{R}[u], \mathfrak{R}[v])]$

Time complexity as claimed in the lemma:

- Just $\mathfrak{L}$ (size $2n-1$) must be proprocessed for RMQ. Total preprocessing: $\mathcal{O}(n) + p(2n-1)$
- For the query: one RMQ in $\mathfrak{L}$ and three constant time array lookups. In total: $\mathcal{O}(1) + q(2n-1)$. □

LCA computed as follows (suppose u is visited before v):

- Nodes between the first visits to u and v: $\mathfrak{D}[\mathfrak{R}[u]..\mathfrak{R}[v]]$
- Shallowest node in this subtour at index $\mathrm{RMQ}_{\mathfrak{L}}(\mathfrak{R}[u], \mathfrak{R}[v])$
- Node at this position and thus output of $\mathrm{LCA}(u, v)$: $\mathfrak{D}[\mathrm{RMQ}_{\mathfrak{L}}(\mathfrak{R}[u], \mathfrak{R}[v])]$

Time complexity as claimed in the lemma:

- Just $\mathfrak{L}$ (size $2n - 1$) must be proprocessed for RMQ. Total preprocessing: $\mathcal{O}(n) + p(2n - 1)$
- For the query: one RMQ in $\mathfrak{L}$ and three constant time array lookups. In total: $\mathcal{O}(1) + q(2n - 1)$. □

LCA computed as follows (suppose u is visited before v):

- Nodes between the first visits to u and v: $\mathfrak{D}[\mathfrak{R}[u]..\mathfrak{R}[v]]$
- Shallowest node in this subtour at index $\text{RMQ}_{\mathfrak{L}}(\mathfrak{R}[u], \mathfrak{R}[v])$
- Node at this position and thus output of $\text{LCA}(u, v)$: $\mathfrak{D}[\text{RMQ}_{\mathfrak{L}}(\mathfrak{R}[u], \mathfrak{R}[v])]$

Time complexity as claimed in the lemma:

- Just $\mathfrak{L}$ (size $2n - 1$) must be proprocessed for RMQ. Total preprocessing: $\mathcal{O}(n) + p(2n - 1)$
- For the query: one RMQ in $\mathfrak{L}$ and three constant time array lookups. In total: $\mathcal{O}(1) + q(2n - 1)$. $\square$

What about RMQ's complexity?

- After procomputing (at least a crucial part of) all possible queries, lookup time $q(n) = \mathcal{O}(1)$.
- Preprocessing time $p(n) =$
  - $\mathcal{O}(n^3)$ – Brute force: For all possible index pairs, search the minimum.
  - $\mathcal{O}(n^2)$ – Still naive: Fill the table by dynamic programming.
  - $\mathcal{O}(n \log n)$ – Better: Precompute only queries for blocks of a power-of-two length; remaining answers may be inferred in constant time at the moment of query.
  - $\mathcal{O}(n)$ – Really clever: Make use of the fact that adjacent elements in $\mathfrak{L}$ differ by exactly ± 1; precompute only solutions for the few generic ± 1-patterns.

What about RMQ's complexity?

- After procomputing (at least a crucial part of) all possible queries, lookup time $q(n) = \mathcal{O}(1)$.
- Preprocessing time $p(n) =$
  - $\mathcal{O}(n^3)$ – Brute force: For all possible index pairs, search the minimum.
  - $\mathcal{O}(n^2)$ – Still naive: Fill the table by dynamic programming.
  - $\mathcal{O}(n \log n)$ – Better: Precompute only queries for blocks of a power-of-two length; remaining answers may be inferred in constant time at the moment of query.
  - $\mathcal{O}(n)$ – Really clever: Make use of the fact that adjacent elements in $\mathfrak{L}$ differ by exactly ± 1; precompute only solutions for the few generic ± 1-patterns.

What about RMQ's complexity?

- After procomputing (at least a crucial part of) all possible queries, lookup time $q(n) = \mathcal{O}(1)$.
- Preprocessing time $p(n) =$
  - $\mathcal{O}(n^3)$ – Brute force: For all possible index pairs, search the minimum.
  - $\mathcal{O}(n^2)$ – Still naive: Fill the table by dynamic programming.
  - $\mathcal{O}(n \log n)$ – Better: Precompute only queries for blocks of a power-of-two length; remaining answers may be inferred in constant time at the moment of query.
  - $\mathcal{O}(n)$ – Really clever: Make use of the fact that adjacent elements in $\mathfrak{L}$ differ by exactly ± 1; precompute only solutions for the few generic ± 1-patterns.

What about RMQ's complexity?

- After procomputing (at least a crucial part of) all possible queries, lookup time $q(n) = \mathcal{O}(1)$.
- Preprocessing time $p(n) =$
  - $\mathcal{O}(n^3)$ – Brute force: For all possible index pairs, search the minimum.
  - $\mathcal{O}(n^2)$ – Still naive: Fill the table by dynamic programming.
  - $\mathcal{O}(n \log n)$ – Better: Precompute only queries for blocks of a power-of-two length; remaining answers may be inferred in constant time at the moment of query.
  - $\mathcal{O}(n)$ – Really clever: Make use of the fact that adjacent elements in $\mathfrak{L}$ differ by exactly ± 1; precompute only solutions for the few generic ± 1-patterns.

What about RMQ's complexity?

- After procomputing (at least a crucial part of) all possible queries, lookup time $q(n) = \mathcal{O}(1)$.
- Preprocessing time $p(n) =$
  - $\mathcal{O}(n^3)$ – Brute force: For all possible index pairs, search the minimum.
  - $\mathcal{O}(n^2)$ – Still naive: Fill the table by dynamic programming.
  - $\mathcal{O}(n \log n)$ – Better: Precompute only queries for blocks of a power-of-two length; remaining answers may be inferred in constant time at the moment of query.
  - $\mathcal{O}(n)$ – Really clever: Make use of the fact that adjacent elements in $\mathfrak{L}$ differ by exactly ± 1; precompute only solutions for the few generic ± 1-patterns.

Edit distance

Definition The *edit distance* (or Levenshtein distance) between two strings S_1 and S_2 is the minimum number of edit operations (insertions, deletions, substitutions) needed to transform S_1 into S_2.

Such a transformation may be coded in an *edit transcript*, i.e. a string over the alphabet $\{I, D, S, M\}$, meaning “insertion”, “deletion”, “substitution” or “match” respectively.

Example

```
RIMDMDMMI
v intner   = S1
wri t ers  = S2
```

Edit distance

Definition The *edit distance* (or Levenshtein distance) between two strings S_1 and S_2 is the minimum number of edit operations (insertions, deletions, substitutions) needed to transform S_1 into S_2.

Such a transformation may be coded in an *edit transcript*, i.e. a string over the alphabet $\{I, D, S, M\}$, meaning "insertion", "deletion", "substitution" or "match" respectively.

Example `RIMDMDMMI`
`v intner  ` $= S_1$
`wri t ers` $= S_2$

Edit distance

Definition The *edit distance* (or Levenshtein distance) between two strings S_1 and S_2 is the minimum number of edit operations (insertions, deletions, substitutions) needed to transform S_1 into S_2.

Such a transformation may be coded in an *edit transcript*, i.e. a string over the alphabet $\{I, D, S, M\}$, meaning “insertion”, “deletion”, “substitution” or “match” respectively.

Example

```
RIMDMDMMI
v intner   = S1
wri t ers  = S2
```

Computing the edit distance

Lemma The edit distance is computable using dynamic programming:

- Build the table $\mathfrak{E}$ where $\mathfrak{E}[i,j]$ denotes the edit distance between $S_1[1..i]$ and $S_2[1..j]$.
- Base conditions: $\mathfrak{E}[i,0] = i$ (all deletions); $\mathfrak{E}[0,j] = j$ (all insertions)
- Recurrence: $\mathfrak{E}[i,j] = \min\{\mathfrak{E}[i,j-1]+1, \mathfrak{E}[i-1,j]+1, \mathfrak{E}[i-1,j-1]+I_{ij}\}$, where $I_{ij} = 0$, if $S_1[i] = S_2[j]$, and $I_{ij} = 1$ otherwise.

Proof The last letter of an optimal transcript is one of $\{I, D, S, M\}$. The recurrence selects the minimum of these possibilities. □

Computing the edit distance

Lemma The edit distance is computable using dynamic programming:

- Build the table $\mathfrak{E}$ where $\mathfrak{E}[i, j]$ denotes the edit distance between $S_1[1..i]$ and $S_2[1..j]$.
- Base conditions: $\mathfrak{E}[i, 0] = i$ (all deletions); $\mathfrak{E}[0, j] = j$ (all insertions)
- Recurrence: $\mathfrak{E}[i, j] = \min\{\mathfrak{E}[i, j-1]+1, \mathfrak{E}[i-1, j]+1, \mathfrak{E}[i-1, j-1]+I_{ij}\}$, where $I_{ij} = 0$, if $S_1[i] = S_2[j]$, and $I_{ij} = 1$ otherwise.

Proof The last letter of an optimal transcript is one of $\{I, D, S, M\}$. The recurrence selects the minimum of these possibilities. □

Computing the edit distance

Lemma The edit distance is computable using dynamic programming:

- Build the table $\mathfrak{E}$ where $\mathfrak{E}[i,j]$ denotes the edit distance between $S_1[1..i]$ and $S_2[1..j]$.
- Base conditions: $\mathfrak{E}[i,0]=i$ (all deletions); $\mathfrak{E}[0,j]=j$ (all insertions)
- Recurrence: $\mathfrak{E}[i,j]=\min\{\mathfrak{E}[i,j-1]+1,\mathfrak{E}[i-1,j]+1,\mathfrak{E}[i-1,j-1]+I_{ij}\}$, where $I_{ij}=0$, if $S_1[i]=S_2[j]$, and $I_{ij}=1$ otherwise.

Proof The last letter of an optimal transcript is one of $\{I,D,S,M\}$. The recurrence selects the minimum of these possibilities. □

Computing the edit distance

Lemma The edit distance is computable using dynamic programming:

- Build the table $\mathfrak{E}$ where $\mathfrak{E}[i,j]$ denotes the edit distance between $S_1[1..i]$ and $S_2[1..j]$.
- Base conditions: $\mathfrak{E}[i,0] = i$ (all deletions); $\mathfrak{E}[0,j] = j$ (all insertions)
- Recurrence:
  $\mathfrak{E}[i,j] = \min\{\mathfrak{E}[i,j-1]+1, \mathfrak{E}[i-1,j]+1, \mathfrak{E}[i-1,j-1]+I_{ij}\}$,
  where $I_{ij} = 0$, if $S_1[i] = S_2[j]$, and $I_{ij} = 1$ otherwise.

Proof The last letter of an optimal transcript is one of $\{I, D, S, M\}$. The recurrence selects the minimum of these possibilities. $\square$

Computing the edit distance

Lemma The edit distance is computable using dynamic programming:

- Build the table $\mathfrak{E}$ where $\mathfrak{E}[i, j]$ denotes the edit distance between $S_1[1..i]$ and $S_2[1..j]$.
- Base conditions: $\mathfrak{E}[i, 0] = i$ (all deletions); $\mathfrak{E}[0, j] = j$ (all insertions)
- Recurrence:
  $\mathfrak{E}[i, j] = \min\{\mathfrak{E}[i, j-1]+1, \mathfrak{E}[i-1, j]+1, \mathfrak{E}[i-1, j-1]+I_{ij}\}$, where $I_{ij} = 0$, if $S_1[i] = S_2[j]$, and $I_{ij} = 1$ otherwise.

Proof The last letter of an optimal transcript is one of $\{I, D, S, M\}$. The recurrence selects the minimum of these possibilities. $\square$

Filling up the table row by row

$\mathfrak{E}[i,j]$	S_2		w	r	i	t	e	r	s
S_1		0	1	2	3	4	5	6	7
	0	0	$\leftarrow$ 1	$\leftarrow$ 2	$\leftarrow$ 3	$\leftarrow$ 4	$\leftarrow$ 5	$\leftarrow$ 6	$\leftarrow$ 7
v	1	$\uparrow$ 1	$\nwarrow$ 1	$\nwarrow\leftarrow$ 2	$\nwarrow\leftarrow$3	$\nwarrow\leftarrow$4	$\nwarrow\leftarrow$5	$\nwarrow\leftarrow$6	$\nwarrow\leftarrow$7
i	2	$\uparrow$ 2	$\nwarrow\leftarrow$2	$\nwarrow$ 2	$\nwarrow$ 2	*			
n	3	$\uparrow$ 3							
t	4	$\uparrow$ 4							
n	5	$\uparrow$ 5							
e	6	$\uparrow$ 6							
r	7	$\uparrow$ 7							

- Complexity: $\mathcal{O}(|S_1| \cdot |S_2|)$
- Note (no proof here): Diagonals are non-decreasing and differ by at most one.

Filling up the table row by row

$\mathfrak{E}[i,j]$	S_2		w	r	i	t	e	r	s
S_1		0	1	2	3	4	5	6	7
	0	0	$\leftarrow 1$	$\leftarrow 2$	$\leftarrow 3$	$\leftarrow 4$	$\leftarrow 5$	$\leftarrow 6$	$\leftarrow 7$
v	1	$\uparrow 1$	$\nwarrow 1$	$\nwarrow\leftarrow 2$	$\nwarrow\leftarrow 3$	$\nwarrow\leftarrow 4$	$\nwarrow\leftarrow 5$	$\nwarrow\leftarrow 6$	$\nwarrow\leftarrow 7$
i	2	$\uparrow 2$	$\nwarrow\leftarrow 2$	$\nwarrow 2$	$\nwarrow 2$	*			
n	3	$\uparrow 3$							
t	4	$\uparrow 4$							
n	5	$\uparrow 5$							
e	6	$\uparrow 6$							
r	7	$\uparrow 7$							

- Complexity: $\mathcal{O}(|S_1| \cdot |S_2|)$
- Note (no proof here): Diagonals are non-decreasing and differ by at most one.

Filling up the table row by row

$\mathfrak{E}[i,j]$	S_2		w	r	i	t	e	r	s
S_1		0	1	2	3	4	5	6	7
	0	0	$\leftarrow$ 1	$\leftarrow$ 2	$\leftarrow$ 3	$\leftarrow$ 4	$\leftarrow$ 5	$\leftarrow$ 6	$\leftarrow$ 7
v	1	$\uparrow$ 1	$\nwarrow$ 1	$\nwarrow\leftarrow$ 2	$\nwarrow\leftarrow$3	$\nwarrow\leftarrow$4	$\nwarrow\leftarrow$5	$\nwarrow\leftarrow$6	$\nwarrow\leftarrow$7
i	2	$\uparrow$ 2	$\nwarrow\leftarrow$2	$\nwarrow$ 2	$\nwarrow$ 2	*			
n	3	$\uparrow$ 3							
t	4	$\uparrow$ 4							
n	5	$\uparrow$ 5							
e	6	$\uparrow$ 6							
r	7	$\uparrow$ 7							

- Complexity: $\mathcal{O}(|S_1| \cdot |S_2|)$
- Note (no proof here): Diagonals are non-decreasing and differ by at most one.

We want some slightly different thing ...

- We need the minimum number of operations to transform $P[1..m]$ so that it *occurs* in $T[1..n]$, not that it actually *is* T; i.e. we want starting spaces to be "free".
- Compute table $\mathfrak{D}$, where

$$\mathfrak{D}[i,j] := \min_{1 \le l \le j} \{\text{edit distance between } P[1..i] \text{ and } T[l..j]\}$$

- Achieved by changing the base conditions: $\mathfrak{D}[i,0] = i$ (as before: all deletions); $\mathfrak{D}[0,j] = 0$ (λ ends anywhere)
- There is a match if row m is reached and if the value there is $\le k$.

We want some slightly different thing ...

- We need the minimum number of operations to transform $P[1..m]$ so that it *occurs* in $T[1..n]$, not that it actually *is* T; i.e. we want starting spaces to be "free".
- Compute table $\mathfrak{D}$, where

$$\mathfrak{D}[i,j] := \min_{1 \leq l \leq j} \{\text{edit distance between } P[1..i] \text{ and } T[l..j]\}$$

- Achieved by changing the base conditions: $\mathfrak{D}[i,0] = i$ (as before: all deletions); $\mathfrak{D}[0,j] = 0$ (λ ends anywhere)
- There is a match if row m is reached and if the value there is $\leq k$.

We want some slightly different thing ...

- We need the minimum number of operations to transform $P[1..m]$ so that it *occurs* in $T[1..n]$, not that it actually *is* T; i.e. we want starting spaces to be "free".
- Compute table $\mathfrak{D}$, where

$$\mathfrak{D}[i,j] := \min_{1 \leq l \leq j} \{\text{edit distance between } P[1..i] \text{ and } T[l..j]\}$$

- Achieved by changing the base conditions: $\mathfrak{D}[i,0] = i$ (as before: all deletions); $\mathfrak{D}[0,j] = 0$ (λ ends anywhere)
- There is a match if row m is reached and if the value there is $\leq k$.

We want some slightly different thing ...

- We need the minimum number of operations to transform $P[1..m]$ so that it *occurs* in $T[1..n]$, not that it actually *is* T; i.e. we want starting spaces to be "free".
- Compute table $\mathfrak{D}$, where

$$\mathfrak{D}[i,j] := \min_{1 \leq l \leq j} \{\text{edit distance between } P[1..i] \text{ and } T[l..j]\}$$

- Achieved by changing the base conditions: $\mathfrak{D}[i,0] = i$ (as before: all deletions); $\mathfrak{D}[0,j] = 0$ (λ ends anywhere)
- There is a match if row m is reached and if the value there is $\leq k$.

Reducing the complexity ...

- ... from $\mathcal{O}(mn)$ to $\mathcal{O}(kn)$ using the *Landau–Vishkin algorithm (LV)*
- Call cell $\mathfrak{D}[i,j]$ an entry of diagonal $j-i$ (range: $-m, ..., n$).
- Do not compute $\mathfrak{D}$ but, column by column, the $(k+1) \times (n+1)$ "meta table" $\mathfrak{L}$ where $\mathfrak{L}[x,y]$ is the row number of the last (i.e. deepest) x along diagonal $y-x$.
- $-k \le y-x \le n$, so all relevant diagonals and thus solutions represented because $\mathfrak{D}[k+1,0] = k+1 > k$ and diagonals are non-decreasing.
- Solution if row m is reached in $\mathfrak{D}$, i.e. if $\mathfrak{L}[x,y] = m$, then there is a match ending at position $m+y-x$ with x differences.

Reducing the complexity ...

- ... from $\mathcal{O}(mn)$ to $\mathcal{O}(kn)$ using the *Landau–Vishkin algorithm (LV)*
- Call cell $\mathfrak{D}[i, j]$ an entry of diagonal $j - i$ (range: $-m, ..., n$).
- Do not compute $\mathfrak{D}$ but, column by column, the $(k + 1) \times (n + 1)$ "meta table" $\mathfrak{L}$ where $\mathfrak{L}[x, y]$ is the row number of the last (i.e. deepest) x along diagonal $y - x$.
- $-k \leq y - x \leq n$, so all relevant diagonals and thus solutions represented because $\mathfrak{D}[k + 1, 0] = k + 1 > k$ and diagonals are non-decreasing.
- Solution if row m is reached in $\mathfrak{D}$, i.e. if $\mathfrak{L}[x, y] = m$; then there is a match ending at position $m + y - x$ with x differences.

Reducing the complexity ...

- ... from $\mathcal{O}(mn)$ to $\mathcal{O}(kn)$ using the *Landau–Vishkin algorithm (LV)*
- Call cell $\mathfrak{D}[i, j]$ an entry of diagonal $j - i$ (range: $-m, ..., n$).
- Do not compute $\mathfrak{D}$ but, column by column, the $(k+1) \times (n+1)$ "meta table" $\mathfrak{L}$ where $\mathfrak{L}[x, y]$ is the row number of the last (i.e. deepest) x along diagonal $y - x$.
- $-k \leq y - x \leq n$, so all relevant diagonals and thus solutions represented because $\mathfrak{D}[k+1, 0] = k+1 > k$ and diagonals are non-decreasing.
- Solution if row m is reached in $\mathfrak{D}$, i.e. if $\mathfrak{L}[x, y] = m$; then there is a match ending at position $m + y - x$ with x differences.

Reducing the complexity ...

- ... from $\mathcal{O}(mn)$ to $\mathcal{O}(kn)$ using the *Landau–Vishkin algorithm (LV)*
- Call cell $\mathfrak{D}[i,j]$ an entry of diagonal $j-i$ (range: $-m, ..., n$).
- Do not compute $\mathfrak{D}$ but, column by column, the $(k+1) \times (n+1)$ "meta table" $\mathfrak{L}$ where $\mathfrak{L}[x,y]$ is the row number of the last (i.e. deepest) x along diagonal $y-x$.
- $-k \leq y-x \leq n$, so all relevant diagonals and thus solutions represented because $\mathfrak{D}[k+1,0] = k+1 > k$ and diagonals are non-decreasing.
- Solution if row m is reached in $\mathfrak{D}$, i.e. if $\mathfrak{L}[x,y] = m$; then there is a match ending at position $m+y-x$ with x differences.

Reducing the complexity ...

- ... from $\mathcal{O}(mn)$ to $\mathcal{O}(kn)$ using the *Landau–Vishkin algorithm (LV)*
- Call cell $\mathfrak{D}[i,j]$ an entry of diagonal $j-i$ (range: $-m, ..., n$).
- Do not compute $\mathfrak{D}$ but, column by column, the $(k+1)\times(n+1)$ "meta table" $\mathfrak{L}$ where $\mathfrak{L}[x,y]$ is the row number of the last (i.e. deepest) x along diagonal $y-x$.
- $-k \leq y-x \leq n$, so all relevant diagonals and thus solutions represented because $\mathfrak{D}[k+1,0]=k+1>k$ and diagonals are non-decreasing.
- Solution if row m is reached in $\mathfrak{D}$, i.e. if $\mathfrak{L}[x,y]=m$; then there is a match ending at position $m+y-x$ with x differences.

How is $\mathfrak{L}$ computed?

- Define $\mathfrak{L}[x, -1] = \mathfrak{L}[x, -2] := -\infty$ because every cell of diagonal $-1 - x$ is at least $\mathfrak{D}[x+1, 0] = x + 1 > x$.
- Fill row 0: $\mathfrak{L}[0, y] = \mathrm{jump}(1, y+1)$, where $\mathrm{jump}(i, j)$ is the longest common prefix of $P[i..m]$ and $T[j..n]$, i.e. $\mathrm{jump}(i, j) = \min\{M_j, \text{length of word } \textsc{lca}(M'_j, \text{leaf } P\$[i..m])\}$

How is $\mathfrak{L}$ computed?

- Define $\mathfrak{L}[x,-1] = \mathfrak{L}[x,-2] := -\infty$ because every cell of diagonal $-1-x$ is at least $\mathfrak{D}[x+1,0] = x+1 > x$.
- Fill row 0: $\mathfrak{L}[0,y] = \text{jump}(1,y+1)$, where $\text{jump}(i,j)$ is the longest common prefix of $P[i..m]$ and $T[j..n]$, i.e.
$\text{jump}(i,j) =$
$\min\{M_j, \text{length of word } \textsc{lca}(M'_j, \text{leaf } P\$[i..m])\}$

Some part of $\mathfrak{L}$:

$$\begin{array}{c|c|c|c|} & \multicolumn{3}{l}{y \longrightarrow} \\ \cline{2-4} x & \alpha & \beta & \gamma \\ \cline{2-4} \downarrow & & & \mathfrak{L}[x,y] \\ \cline{2-4} \end{array}$$

- $\alpha := \mathfrak{L}[x-1, y-2]$ (last $x-1$ on diagonal $y-x-1$)
  $\leftarrow$ insert $T[\alpha + y - x]$ after $P[\alpha]$
- $\beta := \mathfrak{L}[x-1, y-1]$ (last $x-1$ on diagonal $y-x$)
  $\nwarrow$ substitute $T[\beta + 1 + y - x]$ after $P[\beta + 1]$
- $\gamma := \mathfrak{L}[x-1, y]$ (last $x-1$ on diagonal $y-x+1$)
  $\uparrow$ delete $P[\gamma + 1]$
- $t := \max\{\alpha, \beta + 1, \gamma + 1\}$
  $\mathfrak{L}[x, y] = t + \text{jump}(t+1, t+1+y-x)$

Some part of $\mathfrak{L}$:

$$\begin{array}{c|c|c|c|} \multicolumn{1}{c}{} & \multicolumn{3}{l}{y \longrightarrow} \\ \cline{2-4} x & \alpha & \beta & \gamma \\ \cline{2-4} \downarrow & & & \mathfrak{L}[x,y] \\ \cline{2-4} \end{array}$$

- $\alpha := \mathfrak{L}[x-1, y-2]$ (last $x-1$ on diagonal $y-x-1$)
  $\leftarrow$ insert $T[\alpha+y-x]$ after $P[\alpha]$
- $\beta := \mathfrak{L}[x-1, y-1]$ (last $x-1$ on diagonal $y-x$)
  $\nwarrow$ substitute $T[\beta+1+y-x]$ after $P[\beta+1]$
- $\gamma := \mathfrak{L}[x-1, y]$ (last $x-1$ on diagonal $y-x+1$)
  $\uparrow$ delete $P[\gamma+1]$
- $t := \max\{\alpha, \beta+1, \gamma+1\}$
  $\mathfrak{L}[x,y] = t + \mathrm{jump}(t+1, t+1+y-x)$

Some part of $\mathfrak{L}$:

$$\begin{array}{c|c|c|c|} \multicolumn{1}{c}{} & \multicolumn{3}{l}{y \longrightarrow} \\ \cline{2-4} x & \alpha & \beta & \gamma \\ \cline{2-4} \downarrow & & & \mathfrak{L}[x,y] \\ \cline{2-4} \end{array}$$

- $\alpha := \mathfrak{L}[x-1, y-2]$ (last $x-1$ on diagonal $y-x-1$)
  $\leftarrow$ insert $T[\alpha + y - x]$ after $P[\alpha]$
- $\beta := \mathfrak{L}[x-1, y-1]$ (last $x-1$ on diagonal $y-x$)
  $\nwarrow$ substitute $T[\beta + 1 + y - x]$ after $P[\beta + 1]$
- $\gamma := \mathfrak{L}[x-1, y]$ (last $x-1$ on diagonal $y-x+1$)
  $\uparrow$ delete $P[\gamma + 1]$
- $t := \max\{\alpha, \beta + 1, \gamma + 1\}$
  $\mathfrak{L}[x,y] = t + \mathrm{jump}(t+1, t+1+y-x)$

Some part of $\mathfrak{L}$:

$$
\begin{array}{c|c|c|c|}
 & \multicolumn{3}{l}{y \longrightarrow} \\
\cline{2-4}
x & \alpha & \beta & \gamma \\
\cline{2-4}
\downarrow & & & \mathfrak{L}[x,y] \\
\cline{2-4}
\end{array}
$$

- $\alpha := \mathfrak{L}[x-1, y-2]$ (last $x-1$ on diagonal $y-x-1$)
  $\leftarrow$ insert $T[\alpha + y - x]$ after $P[\alpha]$
- $\beta := \mathfrak{L}[x-1, y-1]$ (last $x-1$ on diagonal $y-x$)
  $\nwarrow$ substitute $T[\beta + 1 + y - x]$ after $P[\beta + 1]$
- $\gamma := \mathfrak{L}[x-1, y]$ (last $x-1$ on diagonal $y-x+1$)
  $\uparrow$ delete $P[\gamma + 1]$
- $t := \max\{\alpha, \beta + 1, \gamma + 1\}$
  $\mathfrak{L}[x, y] = t + \mathrm{jump}(t+1, t+1+y-x)$

Now I'm hungry!
Let's go over to ...

Part II

Cooking the Meal

The Algorithm

Linear expected time

Conditions:

1. $T[1..n]$ is a uniformly random string over a b-letter alphabet.
2. Number of differences allowed in a match is

$$k < k^* = \frac{m}{\log_b m + c_1} - c_2.$$

(constants c_i to be specified later; m: pattern length)

Pattern P need not be random.

Linear expected time

Conditions:

1. $T[1..n]$ is a uniformly random string over a b-letter alphabet.
2. Number of differences allowed in a match is

$$k < k^* = \frac{m}{\log_b m + c_1} - c_2.$$

(constants c_i to be specified later; m: pattern length)

Pattern P need not be random.

Linear expected time

Conditions:

1. $T[1..n]$ is a uniformly random string over a b-letter alphabet.
2. Number of differences allowed in a match is

$$k < k^* = \frac{m}{\log_b m + c_1} - c_2.$$

(constants c_i to be specified later; m: pattern length)

Pattern P need not be random.

The Chang–Lawler algorithm (CL)

$s_1 := 1;\ j := 1$

do

  $s_{j+1} := s_j + \mathfrak{M}[s_j] + 1;$ // *compute the start "positions"*

  $j := j + 1$

until $s_j > n$

$j_{max} := j - 1$

for $j := 1$ to j_{max} **do**

  if $(j + k + 2 \leq j_{max}) \wedge (s_{j+k+2} - s_j \leq m - k)$ **then**

    apply LV to $T[s_j..s_{j+k+2} - 1]$ **fi** // *"work at s_j"*

rof

Why does it work?

- If $T[p..p+d-1]$ matches P and $s_j \leq p \leq s_{j+1}$, then this string can be written in the form $\zeta_1 x_1 \zeta_2 x_2 ... \zeta_{k+1} x_{k+1}$, where each x_l is a letter or empty, and each ζ_l is a substring of P.
- Show by induction that, for every $0 \leq l \leq k+1$, $s_{j+l+1} \geq p + \text{length}(\zeta_1 x_1 ... \zeta_l x_l)$. (If you can't live without having seen it, tell me ...)
- So in particular $s_{j+k+2} \geq p + d$, which implies $s_{j+k+2} - s_j \geq d \geq m - k$.
- So CL will perform work at start position s_j and thereby detect there is a match ending at position $p + d - 1$. $\square$

Why does it work?

- If $T[p..p+d-1]$ matches P and $s_j \leq p \leq s_{j+1}$, then this string can be written in the form $\zeta_1 x_1 \zeta_2 x_2 ... \zeta_{k+1} x_{k+1}$, where each x_l is a letter or empty, and each ζ_l is a substring of P.
- Show by induction that, for every $0 \leq l \leq k+1$, $s_{j+l+1} \geq p + \text{length}(\zeta_1 x_1 ... \zeta_l x_l)$. (If you can't live without having seen it, tell me ...)
- So in particular $s_{j+k+2} \geq p + d$, which implies $s_{j+k+2} - s_j \geq d \geq m - k$.
- So CL will perform work at start position s_j and thereby detect there is a match ending at position $p + d - 1$. $\square$

Why does it work?

- If $T[p..p+d-1]$ matches P and $s_j \leq p \leq s_{j+1}$, then this string can be written in the form $\zeta_1 x_1 \zeta_2 x_2 ... \zeta_{k+1} x_{k+1}$, where each x_l is a letter or empty, and each ζ_l is a substring of P.
- Show by induction that, for every $0 \leq l \leq k+1$, $s_{j+l+1} \geq p + \mathrm{length}(\zeta_1 x_1 ... \zeta_l x_l)$. (If you can't live without having seen it, tell me ...)
- So in particular $s_{j+k+2} \geq p + d$, which implies $s_{j+k+2} - s_j \geq d \geq m - k$.
- So CL will perform work at start position s_j and thereby detect there is a match ending at position $p + d - 1$. □

Why does it work?

- If $T[p..p+d-1]$ matches P and $s_j \leq p \leq s_{j+1}$, then this string can be written in the form $\zeta_1 x_1 \zeta_2 x_2 ... \zeta_{k+1} x_{k+1}$, where each x_l is a letter or empty, and each ζ_l is a substring of P.
- Show by induction that, for every $0 \leq l \leq k+1$, $s_{j+l+1} \geq p + \text{length}(\zeta_1 x_1 ... \zeta_l x_l)$. (If you can't live without having seen it, tell me ...)
- So in particular $s_{j+k+2} \geq p + d$, which implies $s_{j+k+2} - s_j \geq d \geq m - k$.
- So CL will perform work at start position s_j and thereby detect there is a match ending at position $p + d - 1$. $\square$

Let's guess what time it is ...

- If we can show the probability to perform work at s_1 is small, this will be true for all s_j's because they are all stochastically independent and equally distributed (because knowledge of all the letters before s_j is of no use when "guessing" s_{j+1}).
- $s_{k^*+3} - s_1 \geq s_{k+3} - s_1$; $m - k \geq m - k^*$
- Thus the event $s_{k+3} - s_1 \geq m - k$ implies the event $s_{k^*+3} - s_1 \geq m - k^*$.
- So $\Pr[s_{k^*+3} - s_1 \geq m - k^*] \geq \Pr[s_{k+3} - s_1 \geq m - k]$ and it suffices to prove the following lemma.

Let's guess what time it is ...

- If we can show the probability to perform work at s_1 is small, this will be true for all s_j's because they are all stochastically independent and equally distributed (because knowledge of all the letters before s_j is of no use when "guessing" s_{j+1}).
- $s_{k^*+3} - s_1 \geq s_{k+3} - s_1$; $m - k \geq m - k^*$
- Thus the event $s_{k+3} - s_1 \geq m - k$ implies the event $s_{k^*+3} - s_1 \geq m - k^*$.
- So $\Pr[s_{k^*+3} - s_1 \geq m - k^*] \geq \Pr[s_{k+3} - s_1 \geq m - k]$ and it suffices to prove the following lemma.

Let's guess what time it is ...

- If we can show the probability to perform work at s_1 is small, this will be true for all s_j's because they are all stochastically independent and equally distributed (because knowledge of all the letters before s_j is of no use when "guessing" s_{j+1}).
- $s_{k^*+3} - s_1 \geq s_{k+3} - s_1$; $m - k \geq m - k^*$
- Thus the event $s_{k+3} - s_1 \geq m - k$ implies the event $s_{k^*+3} - s_1 \geq m - k^*$.
- So $\Pr[s_{k^*+3} - s_1 \geq m - k^*] \geq \Pr[s_{k+3} - s_1 \geq m - k]$ and it suffices to prove the following lemma.

Let's guess what time it is ...

- If we can show the probability to perform work at s_1 is small, this will be true for all s_j's because they are all stochastically independent and equally distributed (because knowledge of all the letters before s_j is of no use when "guessing" s_{j+1}).
- $s_{k^*+3} - s_1 \geq s_{k+3} - s_1$; $m - k \geq m - k^*$
- Thus the event $s_{k+3} - s_1 \geq m - k$ implies the event $s_{k^*+3} - s_1 \geq m - k^*$.
- So $\Pr[s_{k^*+3} - s_1 \geq m - k^*] \geq \Pr[s_{k+3} - s_1 \geq m - k]$ and it suffices to prove the following lemma.

Lemma For suitably chosen constants c_1 and c_2, and $k^* = \frac{m}{\log_b m + c_1} - c_2$, $\Pr[s_{k^*+3} - s_1 \geq m - k^*] < 1/m^3$.

Proof For the sake of easiness, let us assume (i) $b = 2$ ($b > 2$ gives slightly smaller c_i's) and (ii) k^* and $\log m$ are integers ($\log m := \log_2 m$).

- Let X_j be the random variable $s_{j+1} - s_j$.
- Note that $s_{k^*+3} - s_1 = X_1 + ... + X_{k^*+2}$ (telescope sum).
- There are $m2^d$ different strings of length $\log m + d$, but at most m such substrings of P.
- Note that $X_1 = \mathfrak{M}[1] + 1$.
- So

$$\Pr[X_1 = \log m + d + 1] \leq 2^{-d} \quad \text{for all integer } d \geq 0 \qquad (1)$$

Lemma For suitably chosen constants c_1 and c_2, and $k^* = \frac{m}{\log_b m + c_1} - c_2$, $\Pr[s_{k^*+3} - s_1 \geq m - k^*] < 1/m^3$.

Proof For the sake of easiness, let us assume (i) $b = 2$ ($b > 2$ gives slightly smaller c_i's) and (ii) k^* and $\log m$ are integers ($\log m := \log_2 m$).

- Let X_j be the random variable $s_{j+1} - s_j$.
- Note that $s_{k^*+3} - s_1 = X_1 + ... + X_{k^*+2}$ (telescope sum).
- There are $m2^d$ different strings of length $\log m + d$, but at most m such substrings of P.
- Note that $X_1 = \mathfrak{M}[1] + 1$.
- So

$$\Pr[X_1 = \log m + d + 1] < 2^{-d} \quad \text{for all integer } d \geq 0 \qquad (1)$$

Lemma For suitably chosen constants c_1 and c_2, and $k^* = \frac{m}{\log_b m + c_1} - c_2$, $\Pr[s_{k^*+3} - s_1 \geq m - k^*] < 1/m^3$.

Proof For the sake of easiness, let us assume (i) $b = 2$ ($b > 2$ gives slightly smaller c_i's) and (ii) k^* and $\log m$ are integers ($\log m := \log_2 m$).

- Let X_j be the random variable $s_{j+1} - s_j$.
- Note that $s_{k^*+3} - s_1 = X_1 + ... + X_{k^*+2}$ (telescope sum).
- There are $m2^d$ different strings of length $\log m + d$, but at most m such substrings of P.
- Note that $X_1 = \mathfrak{M}[1] + 1$.
- So

$$\Pr[X_1 = \log m + d + 1] < 2^{-d} \quad \text{for all integer } d \geq 0 \quad (1)$$

Lemma For suitably chosen constants c_1 and c_2, and $k^* = \frac{m}{\log_b m + c_1} - c_2$, $\Pr[s_{k^*+3} - s_1 \geq m - k^*] < 1/m^3$.

Proof For the sake of easiness, let us assume (i) $b = 2$ ($b > 2$ gives slightly smaller c_i's) and (ii) k^* and $\log m$ are integers ($\log m := \log_2 m$).

- Let X_j be the random variable $s_{j+1} - s_j$.
- Note that $s_{k^*+3} - s_1 = X_1 + \ldots + X_{k^*+2}$ (telescope sum).
- There are $m2^d$ different strings of length $\log m + d$, but at most m such substrings of P.
- Note that $X_1 = \mathfrak{M}[1] + 1$.
- So

$$\Pr[X_1 = \log m + d + 1] < 2^{-d} \quad \text{for all integer } d \geq 0 \qquad (1)$$

Lemma For suitably chosen constants c_1 and c_2, and $k^* = \frac{m}{\log_b m + c_1} - c_2$, $\Pr[s_{k^*+3} - s_1 \geq m - k^*] < 1/m^3$.

Proof For the sake of easiness, let us assume (i) $b = 2$ ($b > 2$ gives slightly smaller c_i's) and (ii) k^* and $\log m$ are integers ($\log m := \log_2 m$).

- Let X_j be the random variable $s_{j+1} - s_j$.
- Note that $s_{k^*+3} - s_1 = X_1 + \ldots + X_{k^*+2}$ (telescope sum).
- There are $m2^d$ different strings of length $\log m + d$, but at most m such substrings of P.
- Note that $X_1 = \mathfrak{M}[1] + 1$.
- So

$$\Pr[X_1 = \log m + d + 1] < 2^{-d} \quad \text{for all integer } d \geq 0 \qquad (1)$$

Lemma For suitably chosen constants c_1 and c_2, and $k^* = \frac{m}{\log_b m + c_1} - c_2$, $\Pr[s_{k^*+3} - s_1 \geq m - k^*] < 1/m^3$.

Proof For the sake of easiness, let us assume (i) $b = 2$ ($b > 2$ gives slightly smaller c_i's) and (ii) k^* and $\log m$ are integers ($\log m := \log_2 m$).

- Let X_j be the random variable $s_{j+1} - s_j$.
- Note that $s_{k^*+3} - s_1 = X_1 + \ldots + X_{k^*+2}$ (telescope sum).
- There are $m2^d$ different strings of length $\log m + d$, but at most m such substrings of P.
- Note that $X_1 = \mathfrak{M}[1] + 1$.
- So

$$\Pr[X_1 = \log m + d + 1] < 2^{-d} \quad \text{for all integer } d \geq 0 \qquad (1)$$

Lemma For suitably chosen constants c_1 and c_2, and $k^* = \frac{m}{\log_b m + c_1} - c_2$, $\Pr[s_{k^*+3} - s_1 \geq m - k^*] < 1/m^3$.

Proof For the sake of easiness, let us assume (i) $b = 2$ ($b > 2$ gives slightly smaller c_i's) and (ii) k^* and $\log m$ are integers ($\log m := \log_2 m$).

- Let X_j be the random variable $s_{j+1} - s_j$.
- Note that $s_{k^*+3} - s_1 = X_1 + ... + X_{k^*+2}$ (telescope sum).
- There are $m2^d$ different strings of length $\log m + d$, but at most m such substrings of P.
- Note that $X_1 = \mathfrak{M}[1] + 1$.
- So

$$\Pr[X_1 = \log m + d + 1] < 2^{-d} \quad \text{for all integer } d \geq 0 \qquad (1)$$

- $\mathbf{E}[X_j] = \mathbf{E}[X_1] < \log m + 3$ after a few estimations.
- Let $Y_i := X_i - \frac{m-k^*}{k^*+2}$.
- Apply Markov's inequality: $\Pr[X \geq h] \leq \mathbf{E}[X]/h$, for all $h > 0$ $(t > 0)$:

$$\begin{aligned}
\Pr[X_1 + \ldots + X_{k^*+2} \geq m - k^*] &= \Pr[Y_1 + \ldots + Y_{k^*+2} \geq 0] \\
&= \Pr[e^{t(Y_1+\ldots+Y_{k^*+2})} \geq e^{t \cdot 0}] \\
&\leq \mathbf{E}[e^{t(Y_1+\ldots+Y_{k^*+2})}]/1 \\
&= \mathbf{E}[e^{tY_1} \cdot \ldots \cdot e^{tY_{k^*+2}}] \\
&= \mathbf{E}[e^{tY_1}] \cdot \ldots \cdot \mathbf{E}[e^{tY_{k^*+2}}] \\
&= \mathbf{E}[e^{tY_1}]^{k^*+2}
\end{aligned}$$

- $\mathbf{E}[X_j] = \mathbf{E}[X_1] < \log m + 3$ after a few estimations.
- Let $Y_i := X_i - \frac{m-k^*}{k^*+2}$.
- Apply Markov's inequality: $\Pr[X \geq h] \leq \mathbf{E}[X]/h$, for all $h > 0$ ($t > 0$):

$$\begin{aligned}
\Pr[X_1 + \ldots + X_{k^*+2} \geq m - k^*] &= \Pr[Y_1 + \ldots + Y_{k^*+2} \geq 0] \\
&= \Pr[e^{t(Y_1+\ldots+Y_{k^*+2})} \geq e^{t \cdot 0}] \\
&\leq \mathbf{E}[e^{t(Y_1+\ldots+Y_{k^*+2})}]/1 \\
&= \mathbf{E}[e^{tY_1} \cdot \ldots \cdot e^{tY_{k^*+2}}] \\
&= \mathbf{E}[e^{tY_1}] \cdot \ldots \cdot \mathbf{E}[e^{tY_{k^*+2}}] \\
&= \mathbf{E}[e^{tY_1}]^{k^*+2}
\end{aligned}$$

- $\mathbf{E}[X_j] = \mathbf{E}[X_1] < \log m + 3$ after a few estimations.
- Let $Y_i := X_i - \frac{m-k^*}{k^*+2}$.
- Apply Markov's inequality: $\Pr[X \geq h] \leq \mathbf{E}[X]/h$, for all $h > 0$ ($t > 0$):

$$\begin{aligned}
\Pr[X_1 + \ldots + X_{k^*+2} \geq m - k^*] &= \Pr[Y_1 + \ldots + Y_{k^*+2} \geq 0] \\
&= \Pr[e^{t(Y_1 + \ldots + Y_{k^*+2})} \geq e^{t \cdot 0}] \\
&\leq \mathbf{E}[e^{t(Y_1 + \ldots + Y_{k^*+2})}]/1 \\
&= \mathbf{E}[e^{tY_1} \cdot \ldots \cdot e^{tY_{k^*+2}}] \\
&= \mathbf{E}[e^{tY_1}] \cdot \ldots \cdot \mathbf{E}[e^{tY_{k^*+2}}] \\
&= \mathbf{E}[e^{tY_1}]^{k^*+2}
\end{aligned}$$

- $\mathbf{E}[X_j] = \mathbf{E}[X_1] < \log m + 3$ after a few estimations.
- Let $Y_i := X_i - \frac{m-k^*}{k^*+2}$.
- Apply Markov's inequality: $\Pr[X \geq h] \leq \mathbf{E}[X]/h$, for all $h > 0$ $(t > 0)$:

$$\begin{aligned}
\Pr[X_1 + \ldots + X_{k^*+2} \geq m - k^*] &= \Pr[Y_1 + \ldots + Y_{k^*+2} \geq 0] \\
&= \Pr[e^{t(Y_1+\ldots+Y_{k^*+2})} \geq e^{t\cdot 0}] \\
&\leq \mathbf{E}[e^{t(Y_1+\ldots+Y_{k^*+2})}]/1 \\
&= \mathbf{E}[e^{tY_1} \cdot \ldots \cdot e^{tY_{k^*+2}}] \\
&= \mathbf{E}[e^{tY_1}] \cdot \ldots \cdot \mathbf{E}[e^{tY_{k^*+2}}] \\
&= \mathbf{E}[e^{tY_1}]^{k^*+2}
\end{aligned}$$

- $\mathbf{E}[X_j] = \mathbf{E}[X_1] < \log m + 3$ after a few estimations.
- Let $Y_i := X_i - \frac{m-k^*}{k^*+2}$.
- Apply Markov's inequality: $\Pr[X \geq h] \leq \mathbf{E}[X]/h$, for all $h > 0$ $(t > 0)$:

$$\begin{aligned}
\Pr[X_1 + \ldots + X_{k^*+2} \geq m - k^*] &= \Pr[Y_1 + \ldots + Y_{k^*+2} \geq 0] \\
&= \Pr[e^{t(Y_1+\ldots+Y_{k^*+2})} \geq e^{t \cdot 0}] \\
&\leq \mathbf{E}[e^{t(Y_1+\ldots+Y_{k^*+2})}]/1 \\
&= \mathbf{E}[e^{tY_1} \cdot \ldots \cdot e^{tY_{k^*+2}}] \\
&= \mathbf{E}[e^{tY_1}] \cdot \ldots \cdot \mathbf{E}[e^{tY_{k^*+2}}] \\
&= \mathbf{E}[e^{tY_1}]^{k^*+2}
\end{aligned}$$

- $\mathbf{E}[X_j] = \mathbf{E}[X_1] < \log m + 3$ after a few estimations.
- Let $Y_i := X_i - \frac{m-k^*}{k^*+2}$.
- Apply Markov's inequality: $\Pr[X \geq h] \leq \mathbf{E}[X]/h$, for all $h > 0$ $(t > 0)$:

$$\begin{aligned}
\Pr[X_1 + \ldots + X_{k^*+2} \geq m - k^*] &= \Pr[Y_1 + \ldots + Y_{k^*+2} \geq 0] \\
&= \Pr[e^{t(Y_1+\ldots+Y_{k^*+2})} \geq e^{t\cdot 0}] \\
&\leq \mathbf{E}[e^{t(Y_1+\ldots+Y_{k^*+2})}]/1 \\
&= \mathbf{E}[e^{tY_1} \cdot \ldots \cdot e^{tY_{k^*+2}}] \\
&= \mathbf{E}[e^{tY_1}] \cdot \ldots \cdot \mathbf{E}[e^{tY_{k^*+2}}] \\
&= \mathbf{E}[e^{tY_1}]^{k^*+2}
\end{aligned}$$

- $\mathbf{E}[X_j] = \mathbf{E}[X_1] < \log m + 3$ after a few estimations.
- Let $Y_i := X_i - \frac{m-k^*}{k^*+2}$.
- Apply Markov's inequality: $\Pr[X \geq h] \leq \mathbf{E}[X]/h$, for all $h > 0$ $(t > 0)$:

$$
\begin{aligned}
\Pr[X_1 + \ldots + X_{k^*+2} \geq m - k^*] &= \Pr[Y_1 + \ldots + Y_{k^*+2} \geq 0] \\
&= \Pr[e^{t(Y_1+\ldots+Y_{k^*+2})} \geq e^{t\cdot 0}] \\
&\leq \mathbf{E}[e^{t(Y_1+\ldots+Y_{k^*+2})}]/1 \\
&= \mathbf{E}[e^{tY_1} \cdot \ldots \cdot e^{tY_{k^*+2}}] \\
&= \mathbf{E}[e^{tY_1}] \cdot \ldots \cdot \mathbf{E}[e^{tY_{k^*+2}}] \\
&= \mathbf{E}[e^{tY_1}]^{k^*+2}
\end{aligned}
$$

- $\mathbf{E}[X_j] = \mathbf{E}[X_1] < \log m + 3$ after a few estimations.
- Let $Y_i := X_i - \frac{m-k^*}{k^*+2}$.
- Apply Markov's inequality: $\Pr[X \geq h] \leq \mathbf{E}[X]/h$, for all $h > 0$ $(t > 0)$:

$$
\begin{aligned}
\Pr[X_1 + \ldots + X_{k^*+2} \geq m - k^*] &= \Pr[Y_1 + \ldots + Y_{k^*+2} \geq 0] \\
&= \Pr[e^{t(Y_1+\ldots+Y_{k^*+2})} \geq e^{t \cdot 0}] \\
&\leq \mathbf{E}[e^{t(Y_1+\ldots+Y_{k^*+2})}]/1 \\
&= \mathbf{E}[e^{tY_1} \cdot \ldots \cdot e^{tY_{k^*+2}}] \\
&= \mathbf{E}[e^{tY_1}] \cdot \ldots \cdot \mathbf{E}[e^{tY_{k^*+2}}] \\
&= \mathbf{E}[e^{tY_1}]^{k^*+2}
\end{aligned}
$$

- $\mathbf{E}[X_j] = \mathbf{E}[X_1] < \log m + 3$ after a few estimations.
- Let $Y_i := X_i - \frac{m-k^*}{k^*+2}$.
- Apply Markov's inequality: $\Pr[X \geq h] \leq \mathbf{E}[X]/h$, for all $h > 0$ ($t > 0$):

$$
\begin{aligned}
\Pr[X_1 + \ldots + X_{k^*+2} \geq m - k^*] &= \Pr[Y_1 + \ldots + Y_{k^*+2} \geq 0] \\
&= \Pr[e^{t(Y_1+\ldots+Y_{k^*+2})} \geq e^{t\cdot 0}] \\
&\leq \mathbf{E}[e^{t(Y_1+\ldots+Y_{k^*+2})}]/1 \\
&= \mathbf{E}[e^{tY_1} \cdot \ldots \cdot e^{tY_{k^*+2}}] \\
&= \mathbf{E}[e^{tY_1}] \cdot \ldots \cdot \mathbf{E}[e^{tY_{k^*+2}}] \\
&= \mathbf{E}[e^{tY_1}]^{k^*+2}
\end{aligned}
$$

- Inequality (1): $\Pr[X_1 = \log m + d + 1] < 2^{-d}$, is equivalent to

  $\Pr[Y_1 = \log m + d + 1 - \frac{m-k^*}{k^*+2}] < 2^{-d}$ for all integer $d \geq 0$

- So, the theorem of total expectation implies, for all $t > 0$ ($\alpha := \log m + 1 - \frac{m-k^*}{k^*+2}$),

$$\begin{aligned}
\mathbf{E}[e^{tY_1}] &= \mathbf{E}[e^{tY_1} | Y_1 \leq \alpha] \cdot \underbrace{\Pr[Y_1 \leq \alpha]}_{\leq 1} + \\
&\quad + \sum_{d=1}^{\infty} \mathbf{E}[e^{tY_1} | Y_1 = \alpha + d] \cdot \Pr[Y_1 = \alpha + d] \\
&\leq e^{t\alpha} + \sum_{d=1}^{\infty} e^{t(\alpha+d)} \cdot \Pr[Y_1 = \alpha + d] \\
&< \sum_{d=0}^{\infty} e^{t(\alpha+d)} \cdot 2^{-d}
\end{aligned}$$

- Inequality (1): $\Pr[X_1 = \log m + d + 1] < 2^{-d}$, is equivalent to

$$\Pr[Y_1 = \log m + d + 1 - \frac{m-k^*}{k^*+2}] < 2^{-d} \quad \text{for all integer } d \geq 0$$

- So, the theorem of total expectation implies, for all $t > 0$ $(\alpha := \log m + 1 - \frac{m-k^*}{k^*+2})$,

$$\begin{aligned}
\mathbf{E}[e^{tY_1}] &= \mathbf{E}[e^{tY_1} | Y_1 \leq \alpha] \cdot \underbrace{\Pr[Y_1 \leq \alpha]}_{\leq 1} + \\
&\quad + \sum_{d=1}^{\infty} \mathbf{E}[e^{tY_1} | Y_1 = \alpha + d] \cdot \Pr[Y_1 = \alpha + d] \\
&\leq e^{t\alpha} + \sum_{d=1}^{\infty} e^{t(\alpha+d)} \cdot \Pr[Y_1 = \alpha + d] \\
&< \sum_{d=0}^{\infty} e^{t(\alpha+d)} \cdot 2^{-d}
\end{aligned}$$

- Inequality (1): $\Pr[X_1 = \log m + d + 1] < 2^{-d}$, is equivalent to

$$\Pr[Y_1 = \log m + d + 1 - \frac{m - k^*}{k^* + 2}] < 2^{-d} \quad \text{for all integer } d \geq 0$$

- So, the theorem of total expectation implies, for all $t > 0$ ($\alpha := \log m + 1 - \frac{m-k^*}{k^*+2}$),

$$\begin{aligned} \mathbf{E}[e^{tY_1}] &= \mathbf{E}[e^{tY_1} | Y_1 \leq \alpha] \cdot \underbrace{\Pr[Y_1 \leq \alpha]}_{\leq 1} + \\ &\quad + \sum_{d=1}^{\infty} \mathbf{E}[e^{tY_1} | Y_1 = \alpha + d] \cdot \Pr[Y_1 = \alpha + d] \\ &\leq e^{t\alpha} + \sum_{d=1}^{\infty} e^{t(\alpha+d)} \cdot \Pr[Y_1 = \alpha + d] \\ &< \sum_{d=0}^{\infty} e^{t(\alpha+d)} \cdot 2^{-d} \end{aligned}$$

- Inequality (1): $\Pr[X_1 = \log m + d + 1] < 2^{-d}$, is equivalent to
$$\Pr[Y_1 = \log m + d + 1 - \frac{m-k^*}{k^*+2}] < 2^{-d} \quad \text{for all integer } d \geq 0$$
- So, the theorem of total expectation implies, for all $t > 0$ ($\alpha := \log m + 1 - \frac{m-k^*}{k^*+2}$),

$$\begin{aligned}
\mathbf{E}[e^{tY_1}] &= \mathbf{E}[e^{tY_1}|Y_1 \leq \alpha] \cdot \underbrace{\Pr[Y_1 \leq \alpha]}_{\leq 1} + \\
&\quad + \sum_{d=1}^{\infty} \mathbf{E}[e^{tY_1}|Y_1 = \alpha + d] \cdot \Pr[Y_1 = \alpha + d] \\
&\leq e^{t\alpha} + \sum_{d=1}^{\infty} e^{t(\alpha+d)} \cdot \Pr[Y_1 = \alpha + d] \\
&< \sum_{d=0}^{\infty} e^{t(\alpha+d)} \cdot 2^{-d}
\end{aligned}$$

Homework Choose $t = \frac{\log_e 2}{2}$, do some algebra, and verify that the following is true for the probability to perform work at position s_1 and thus at each position:

$$\begin{aligned} \Pr[s_{k^*+3} - s_1 \geq m - k^*] &\leq \mathbf{E}[e^{tY_1}]^{k^*+2} \\ &< (\sum_{d=0}^{\infty} e^{t(\alpha+d)} \cdot 2^{-d})^{k^*+2} \\ &<^! 1/m^3, \end{aligned}$$

if $c_1 = 5.6$ and $c_2 = 8$.

So what time is it?

LV is applied with a probability of less than $1/m^3$, the text it is applied to is supposed to have length $(k+2)\mathbf{E}[X_1] < (k+2)(\log m + 3) = \mathcal{O}(k \log m)$, and LV has complexity $\mathcal{O}(kl)$, if l is the length of the input string.

Also recall that $k = \mathcal{O}(\frac{m}{\log m})$.

So the average expected work for any start position s_j is

$$\begin{aligned} m^{-3}\mathcal{O}(k^2 \log m) &= m^{-3}\mathcal{O}(\frac{m^2}{(\log m)^2} \log m) \\ &= \mathcal{O}(\frac{1}{m \log m}) \\ &= \mathcal{O}(\lambda n.\lambda m.1) \end{aligned}$$

Hence the total expected work is $\mathcal{O}(n)$. □

So what time is it?

LV is applied with a probability of less than $1/m^3$, the text it is applied to is supposed to have length $(k+2)\mathbf{E}[X_1] < (k+2)(\log m + 3) = \mathcal{O}(k \log m)$, and LV has complexity $\mathcal{O}(kl)$, if l is the length of the input string.
Also recall that $k = \mathcal{O}(\frac{m}{\log m})$.
So the average expected work for any start position s_j is

$$\begin{aligned} m^{-3}\mathcal{O}(k^2 \log m) &= m^{-3}\mathcal{O}(\frac{m^2}{(\log m)^2} \log m) \\ &= \mathcal{O}(\frac{1}{m \log m}) \\ &= \mathcal{O}(\lambda n.\lambda m.1) \end{aligned}$$

Hence the total expected work is $\mathcal{O}(n)$. □

So what time is it?

LV is applied with a probability of less than $1/m^3$, the text it is applied to is supposed to have length $(k+2)\mathbf{E}[X_1] < (k+2)(\log m + 3) = \mathcal{O}(k \log m)$, and LV has complexity $\mathcal{O}(kl)$, if l is the length of the input string.
Also recall that $k = \mathcal{O}(\frac{m}{\log m})$.
So the average expected work for any start position s_j is

$$
\begin{aligned}
m^{-3}\mathcal{O}(k^2 \log m) &= m^{-3}\mathcal{O}(\frac{m^2}{(\log m)^2} \log m) \\
&= \mathcal{O}(\frac{1}{m \log m}) \\
&= \mathcal{O}(\lambda n.\lambda m.1)
\end{aligned}
$$

Hence the total expected work is $\mathcal{O}(n)$. □

So what time is it?

LV is applied with a probability of less than $1/m^3$, the text it is applied to is supposed to have length $(k+2)\mathbf{E}[X_1] < (k+2)(\log m + 3) = \mathcal{O}(k \log m)$, and LV has complexity $\mathcal{O}(kl)$, if l is the length of the input string.
Also recall that $k = \mathcal{O}(\frac{m}{\log m})$.
So the average expected work for any start position s_j is

$$
\begin{aligned}
m^{-3}\mathcal{O}(k^2 \log m) &= m^{-3}\mathcal{O}(\frac{m^2}{(\log m)^2} \log m) \\
&= \mathcal{O}(\frac{1}{m \log m}) \\
&= \mathcal{O}(\lambda n.\lambda m.1)
\end{aligned}
$$

Hence the total expected work is $\mathcal{O}(n)$. □

So what time is it?

LV is applied with a probability of less than $1/m^3$, the text it is applied to is supposed to have length $(k+2)\mathbf{E}[X_1] < (k+2)(\log m + 3) = \mathcal{O}(k \log m)$, and LV has complexity $\mathcal{O}(kl)$, if l is the length of the input string.
Also recall that $k = \mathcal{O}(\frac{m}{\log m})$.
So the average expected work for any start position s_j is

$$
\begin{aligned}
m^{-3}\mathcal{O}(k^2 \log m) &= m^{-3}\mathcal{O}(\frac{m^2}{(\log m)^2} \log m) \\
&= \mathcal{O}(\frac{1}{m \log m}) \\
&= \mathcal{O}(\lambda n.\lambda m.1)
\end{aligned}
$$

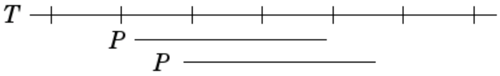
Hence the total expected work is $\mathcal{O}(n)$. $\square$

Let's go beneath the line: SET

Now an algorithm is derived from LET that is sublinear in n (when $k < k^*/2 - 3$; k^* as before).

The trick is:

- Partition T into regions of length $\frac{m-k}{2}$.
  Any substring of T that matches P must contain the whole of at least one region:



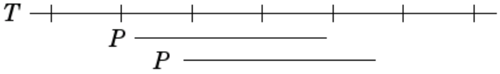
- Starting from the left end of each region R, compute $k+1$ "maximum jumps" (using $\mathfrak{M}$), say ending at position p.
  If p is within R, there can be no match containing the whole of R.
  If p is beyond R, apply LV to a stretch of text beginning $\frac{m+3k}{2}$ letters to the left of R and ending at p.

Let's go beneath the line: SET

Now an algorithm is derived from LET that is sublinear in n (when $k < k^*/2 - 3$; k^* as before).

The trick is:

- Partition T into regions of length $\frac{m-k}{2}$.
  Any substring of T that matches P must contain the whole of at least one region:



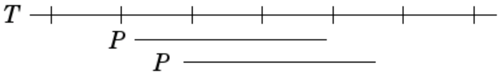
- Starting from the left end of each region R, compute $k+1$ "maximum jumps" (using $\mathfrak{M}$), say ending at position p.
  If p is within R, there can be no match containing the whole of R.
  If p is beyond R, apply LV to a stretch of text beginning $\frac{m+3k}{2}$ letters to the left of R and ending at p.

Let's go beneath the line: SET

Now an algorithm is derived from LET that is sublinear in n (when $k < k^*/2 - 3$; k^* as before).

The trick is:

- Partition T into regions of length $\frac{m-k}{2}$.
  Any substring of T that matches P must contain the whole of at least one region:



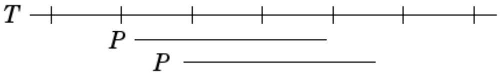
- Starting from the left end of each region R, compute $k+1$ "maximum jumps" (using $\mathfrak{M}$), say ending at position p.
  If p is within R, there can be no match containing the whole of R.
  If p is beyond R, apply LV to a stretch of text beginning $\frac{m+3k}{2}$ letters to the left of R and ending at p.

Let's go beneath the line: SET

Now an algorithm is derived from LET that is sublinear in n (when $k < k^*/2 - 3$; k^* as before).

The trick is:

- Partition T into regions of length $\frac{m-k}{2}$.
  Any substring of T that matches P must contain the whole of at least one region:



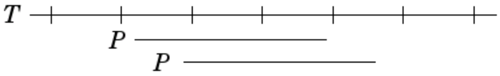
- Starting from the left end of each region R, compute $k+1$ "maximum jumps" (using $\mathfrak{M}$), say ending at position p.
  If p is within R, there can be no match containing the whole of R.
  If p is beyond R, apply LV to a stretch of text beginning $\frac{m+3k}{2}$ letters to the left of R and ending at p.

Let's go beneath the line: SET

Now an algorithm is derived from LET that is sublinear in n (when $k < k^*/2 - 3$; k^* as before).

The trick is:

- Partition T into regions of length $\frac{m-k}{2}$.
  Any substring of T that matches P must contain the whole of at least one region:



- Starting from the left end of each region R, compute $k+1$ "maximum jumps" (using $\mathfrak{M}$), say ending at position p.
  If p is within R, there can be no match containing the whole of R.
  If p is beyond R, apply LV to a stretch of text beginning $\frac{m+3k}{2}$ letters to the left of R and ending at p.

- A variation of the proof for LET yields that

$$\Pr[p \text{ is beyond } R] < 1/m^3$$

- So, similarly to the analysis of LET, the total expected work is:

$$m^{-3} \underbrace{\frac{2n}{m-k}}_{\sharp \text{ regions}} \underbrace{[(k+1)(\log m + \mathcal{O}(1)) + \mathcal{O}(m)]}_{\text{exp. work at region examined}} = \ldots = \mathcal{O}(n/m^3)$$

$$= o(n)$$

□

- A variation of the proof for LET yields that

$$\Pr[p \text{ is beyond } R] < 1/m^3$$

- So, similarly to the analysis of LET, the total expected work is:

$$m^{-3} \underbrace{\frac{2n}{m-k}}_{\sharp \text{ regions}} \underbrace{[(k+1)(\log m + \mathcal{O}(1)) + \mathcal{O}(m)]}_{\text{exp. work at region examined}} = \ldots = \mathcal{O}(n/m^3)$$

$$= o(n)$$

$\square$

- A variation of the proof for LET yields that

$$\Pr[p \text{ is beyond } R] < 1/m^3$$

- So, similarly to the analysis of LET, the total expected work is:

$$m^{-3} \underbrace{\frac{2n}{m-k}}_{\sharp \text{ regions}} \underbrace{[(k+1)(\log m + \mathcal{O}(1)) + \mathcal{O}(m)]}_{\text{exp. work at region examined}} = \ldots = \mathcal{O}(n/m^3)$$

$$= o(n)$$

$\square$

At last some practical notes

- A combination of LET (for $k \geq k^*/2 - 3$) and SET (for $k < k^*/2 - 3$) runs in $\mathcal{O}(\frac{n}{m} k \log m)$ expected time.
- In a 16-letter alphabet, k^* may be up to 25% of m, in a 64-letter alphabet even 35%.

At last some practical notes

- A combination of LET (for $k \geq k^*/2 - 3$) and SET (for $k < k^*/2 - 3$) runs in $\mathcal{O}(\frac{n}{m} k \log m)$ expected time.
- In a 16-letter alphabet, k^* may be up to 25% of m, in a 64-letter alphabet even 35%.

The moral

Mind the preprocessing!

“Gut gekaut ist halb verdaut.”
“A good chewing is half the digestion.”