Chapter 2

Sublinear Approximate String Matching

Robert West

The present paper deals with the subject of approximate string matching and demonstrates how Chang and Lawler [CL94] conceived a new sublinear time algorithm out of ideas that had previously been known.
The problem is to find all locations in a text of length n over a b-letter alphabet where a pattern of length m occurs with up to k differences (substitutions, insertions, deletions).
The algorithm will run in $\mathcal{O}(\frac{n}{m}k\log_b m)$ time when the text is random and k is bounded by the threshold $m/(\log_b m + \mathcal{O}(1))$. In particular, when $k = o(m/\log_b m)$ the expected running time is $o(n)$.

2.1 Introduction

I never have found the perfect quote. At best I have been able to find a *string* of quotations which merely circle the ineffible idea I seek to express.

Caldwell O'Keefe

2.1.1 What?

In order to be able to explain why and how we will solve a problem, we are bound to exactly define what the problem is.

Definition 2.1. 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).

The following examply will clarify the matter:

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

"YELTSIN" matches in "TORTELLINI" with three differences: The 'Y' in "YELTSIN" is replaced by a 'T', the 'T' in "YELTSIN" is deleted and the 'S' in "YELTSIN" is replaced by an 'L'.

2.1.2 Why?

Approximate string matching as a generalisation of exact string matching has been made necessary due to one major reason:
Genetics is the science that has, in the last years, conjured up a set of new challenges in the field of string processing. (e.g. the search for a string like "GCACTT..." in a huge gene database)
Sequencing techniques, however, are not perfect: The experimental error is up to 5–10%.
Moreover, gene mutation (leading to polymorphism; cf. Darwin's ideas) is a *condicio sine qua non*, it is the mother of evolution. Thus matching a piece of DNA against a database of many individuals must allow a small but significant error.

2.1.3 How?

We will first gather the ingredients (suffix trees, matching statistics, lowest common ancestor retrieval, edit distance) and then merge the ingredients to form the algorithm: We will develop the linear expected time algorithm (LET) in detail and will then obtain the sublinear expected time algorithm (SET) after some modifications.
We will mainly stick to the paper of Chang and Lawler [CL94]; since it is very concise I added material from further sources where it seemed necessary for reasons of understanding.

2.2 The Auxiliary Tools

> Hunger, work and sweat are the best spices.
>
> *Icelandic proverb*

– Not in our case! Our spices are suffix trees, matching statistics, lowest common ancestor retrieval and edit distance. (An apt and detailed introduction to all of these concepts can be found in [Gus97].)

2.2.1 Suffix Trees

Suffix trees may be called *the* data structure for all sorts of string matching. Thus it takes small wonder that they will extensively be used in our case, too.
We will denote the suffix tree of a string $P[1..m]\$$ ($\$$ is the terminal symbol that appears nowhere but at the last position) as $\mathfrak{S}_P$.
A string α is called a *branching word* (of $P\$$) iff there are different letters x and

y such that both αx and αy are substrings of $P\$$.
Quite obviously, the following correlations hold:

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

Next we define floor(α) to be the longest prefix of α that is a branching word and ceil(α) to be the shortest extension of α that is a branching word or a suffix of $P\$$. Note that α is a branching word iff $\text{floor}(\alpha) = \text{ceil}(\alpha) = \alpha$.
To make matters simpler, we introduce the 'inverted string': Let $\beta^{-1}\alpha$ be the string α without its prefix β; of course this only makes sense, if β really is a prefix of α.
The nodes of the tree will be labelled with the correlating branching words or suffixes respectively, while an edge (β, α) will be labelled with the triple (x, l, r) such that $P\$[l] = x$ and $\beta^{-1}\alpha = P\$[l..r]$.
The following are intuitionally clear: $\text{son}(\beta, x) := \alpha$; $\text{len}(\beta, x) := r - l + 1$. Furthermore, let $\text{first}(\beta, x) := l$ (the position of the first letter in $P\$$, not the letter itself, which is already known to be x).
At last, let shift(α) be α without its first letter (if $\alpha \neq \varepsilon$); so applying the shift function means following a suffix link (cf. [Ukk95, Gus97]).

2.2.2 Matching Statistics

The next crucial data structure to be used is matching statistics. It will store information about the occurrence of a pattern in a text; this is put more precisely by the following

Definition 2.2. 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 $\mathfrak{M}'$.

Our goal is to find an $\mathcal{O}(n+m)$ time algorithm for computing the matching statistics of T and P in a single left-to-right scan of T using just $\mathfrak{S}_P$. (We will follow [CL94].)
Brief description of the algorithm: 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", [Ukk95, Gus97])

What follows now is some notes that shall serve to clarify the pseudo-code given later:

- 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 $\mathcal{O}(m)$ for steps 1 and 2 (use e.g. Ukkonen's [Ukk95, Gus97] or McCreight's [Gus97] algorithm to construct the suffix tree), yielding together the desired $\mathcal{O}(n + m)$.

After the above explanations the following code, which computes the matching statistics of T with respect to P, should be more easily understandable:

```
1     construct 𝔖_P in 𝒪(m) time
2     α := root; j := k := 1
3     for i := 1 to n do
3.1       while (j < k) ∧ (j + len(α, T[j]) ≤ k) do    // "skip and count"
              α := son(α, T[j]);
              j := j + len(α, T[j])
          elihw
3.2       if j = k then    // extend the match
              while son(α, T[j]) exists ∧ T[k] = P$[first(α, T[j]) + k − j] do
                  k := k + 1
                  if k = j + len(α, T[j]) then
                      α := son(α, T[j]);
                      j := k fi
              elihw
          fi
3.3       𝔐[i] := k − i
          if j = k then 𝔐′[i] := α
              else 𝔐′[i] := son(α, T[j]) fi
3.4       if (α is root) ∧ (j = k) then
              j := j + 1;
              k := k + 1 fi
          if (α is root) ∧ (j < k) then
              j := j + 1 fi
          if (α is not root) then
              α := shift(α) fi
      rof
```

2.2.3 Lowest Common Ancestor

Having introduced the notion of suffix trees, we will at some points while deriving the main algorithm be interested in the node that is an ancestor to two given nodes of the tree and that is 'minimal' with that quality, meaning it is the node furthest from the root. More formally:

Definition 2.3. For nodes u, v of a rooted tree $\mathfrak{T}$, $\textsc{lca}(u, v)$ is the node furthest from the root that is an ancestor to both u and v.

Our goal is a constant time LCA retrieval after some preprocessing; it is achieved by reducing the LCA problem to the *range minimum query (RMQ)* problem as proposed in [BFC00]. RMQ operates on arrays and is defined as follows.

Definition 2.4. For an array $\mathfrak{A}$ and indices i and j, $\textsc{rmq}_{\mathfrak{A}}(i, j)$ is the index of the smallest element in the subarray $\mathfrak{A}[i..j]$.

For ease in notation we will say that, if an algorithm has preprocessing time $p(n)$ and query time $q(n)$, it has complexity $\langle p(n), q(n)\rangle$.
The following main lemma states how closely the complexities of RMQ and LCA are connected.

Lemma 2.1. *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: $1, ..., n$).

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 $\mathfrak{R}[w] = \min\{j \mid \mathfrak{D}[j] = w\}$.

All of this is feasible during a single DFS; thus the running time is $\mathcal{O}(n)$.
Now the LCA may be computed as follows (suppose u is visited before v): The nodes between the first visits to u and v are stored in $\mathfrak{D}[\mathfrak{R}[u]..\mathfrak{R}[v]]$. The shallowest node in this subtour is found at index $\text{RMQ}_{\mathfrak{L}}(\mathfrak{R}[u], \mathfrak{R}[v])$. The node at this position and thus the output of $\text{LCA}(u, v)$ is $\mathfrak{D}[\text{RMQ}_{\mathfrak{L}}(\mathfrak{R}[u], \mathfrak{R}[v])]$.
The time complexity is as claimed in the lemma: Just $\mathfrak{L}$ (size $2n-1$) must be proprocessed for RMQ. So the total preprocessing time is $\mathcal{O}(n) + p(2n-1)$. For the query one RMQ in $\mathfrak{L}$ and three constant time array lookups are needed. In total we get $\mathcal{O}(1) + q(2n-1)$ query time. □

What about RMQ's complexity? – After procomputing (at least a crucial part of) all possible queries, lookup time is $q(n) = \mathcal{O}(1)$.
Preprocessing time $p(n)$ is $\mathcal{O}(n^3)$ by a brute force algorithm: For all possible index pairs, search the minimum.
It is $\mathcal{O}(n^2)$ if we fill the table by dynamic programming. This is, however, still naive.
A far better algorithm runs in $\mathcal{O}(n \log n)$ time: Precompute only queries for blocks of a power-of-two length; remaining answers may then be inferred in constant time at the moment of query.
Finally, a really clever $\mathcal{O}(n)$ time solution was proposed by Bender and Colton: It makes use of the fact that adjacent elements in $\mathfrak{L}$ differ by exactly ± 1; so only solutions for the few generic ± 1-patterns are precomputed. It can be shown that there are only $\mathcal{O}(\sqrt{n})$ such patterns and that preprocessing is then $\mathcal{O}(\sqrt{n}(\log n)^2) = \mathcal{O}(n)$.
For details regarding the two latter cases, I refer to [BFC00]; dwelling upon them would prolong this report in an undue way.

2.2.4 Edit Distance

We will now introduce another data structure called 'edit distance', which will be needed in the main algorithm:

Definition 2.5. 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. The following example will clarify this (S_1 is to be transformed to S_2). Note there may be more than one transformation – even more than one optimal transformation.

Example
```
SIMDMDMMI
v intner   = S1
wri t ers  = S2
```

In the now to be presented 'naive' dynamic programming solution we need not use the auxiliary tools already gathered (suffix trees, matching statistics, LCA), but later on – in the Landau–Vishkin algorithm – we will do so to obtain some lower complexity.

Lemma 2.2. *The edit distance is computable using dynamic programming.*

Proof. What we want to do is to build the table $\mathfrak{E}$ where $\mathfrak{E}[i,j]$ denotes the edit distance between $S_1[1..i]$ and $S_2[1..j]$. Then $\mathfrak{E}[n_1,n_2]$ will be the edit distance between the complete strings $S_1[1..n_1]$ and $S_2[1..n_2]$.
The base conditions are: $\mathfrak{E}[i,0] = i$ (all deletions); $\mathfrak{E}[0,j] = j$ (all insertions)
The recurrence scheme is:

$$\mathfrak{E}[i,j] = \min\{\mathfrak{E}[i,j-1]+1, \mathfrak{E}[i-1,j]+1, \mathfrak{E}[i-1,j-1]+I_{ij}\},$$

where $I_{ij} = 0$, if $S_1[i] = S_2[j]$, and $I_{ij} = 1$ otherwise.

The last letter of an optimal transcript is one of $\{I, D, S, M\}$. The recurrence selects the minimum of these possibilities. For example, if the last letter of an optimal transcript is I then $S_2[n_2]$ is appended to the end of S_1, and what remains to be done is to transform S_1 to $S_2[1..n_2-1]$. The edit distance between these two strings is already known, and the first of the three possibilities is chosen. □

Here is some part of the table for the example given above (The arrows point to the cells from which a particular cell may be computed and thus represent one of $\{I, D, S, M\}$). That there is often more than one arrow makes clear once more that there may be more than one optimal way to transform one string into another. Each trace that follows a sequence of arrows from cell (n_1, n_2) to cell $(1,1)$ represents an optimal transcript.

$\mathfrak{E}[i,j]$	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	$\uparrow 7$							

The complexity is $\mathcal{O}(|S_1| \cdot |S_2|)$.
Also note that rows, columns and diagonals are non-decreasing and differ by at most one.

We need, however, 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".
Thus we need a table $\mathfrak{D}$, where

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

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

It is possible to reduce the complexity from $\mathcal{O}(mn)$ to $\mathcal{O}(kn)$; the method is called the *Landau–Vishkin algorithm (LV)*:

Call cell $\mathfrak{D}[i,j]$ an entry of diagonal $j-i$ (range: $-m, ..., n$).
LV makes use of one fact: When computing $\mathfrak{D}$, we may not omit whole rows or whole columns, but we may in fact omit whole diagonals. That is why we will not compute $\mathfrak{D}$ but, column by column, the $(k+1)\times(n+1)$ "meta table" $\mathfrak{L}$ where $\mathfrak{L}[x,y]$ (with x ranging from 0 to k and y from 0 to n) 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 are represented because $\mathfrak{D}[k+1,0] = k+1 > k$ and diagonals are non-decreasing.
There is a 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.

First, define $\mathfrak{L}[x,-1] = \mathfrak{L}[x,-2] := -\infty$; this is sensible because every cell of diagonal $-1-x$ is at least $\mathfrak{D}[x+1,0] = x+1 > x$.
Now fill row 0: $\mathfrak{L}[0,y] = \text{jump}(1, y+1)$, where jump$(i,j)$ is the longest common prefix of $P[i..m]$ and $T[j..n]$, i.e.

$$\text{jump}(i,j) := \min\{\mathfrak{M}[j], \text{length of word } \textsc{lca}(\mathfrak{M}'[j], \text{leaf } P\$[i..m])\}.$$

Consider some part of $\mathfrak{L}$:

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

$\alpha = \mathfrak{L}[x-1, y-2]$, $\beta = \mathfrak{L}[x-1, y-1]$ and $\gamma = \mathfrak{L}[x-1, y]$ denote respectively the row numbers of the last $x-1$ on diagonals $y-x-1$, $y-x$ and $y-x+1$. Along diagonal $y-x$ the cells at rows α, $\beta+1$ and $\gamma+1$ are at most x (due to the non-decreasingness and difference by at most one). The cell in row $\beta+1$ must be exactly x. So it follows from the non-decreasingness of diagonals that the deepest of these three cells (in row $t := \max\{\alpha, \beta+1, \gamma+1\}$) must have value x, too. To find the row where the *last* x occurs on diagonal $y-x$, we go down this diagonal as long as the value does not increase to $x+1$. Thus cell (x,y) may be computed as follows:

$$\mathfrak{L}[x,y] = t + \text{jump}(t+1, t+1+y-x)$$

2.3 The Algorithm

Virtus in usu sui tota posita est.

Marcus Tullius Cicero, De re publica

– "The value of virtue is entirely in its use." So let us apply all the "virtual" assets we have gathered up to now and make use of them in the core algorithm.

2.3.1 Linear Expected Time

The algorithm me develop first will run in $\mathcal{O}(n)$ time when the following two conditions hold:

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

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

(the constants c_i are to be specified later; m is the pattern length)

Pattern P need not be random.

The algorithm is named after those who invented it – Chang and Lawler (CL) – and is given in pseudo-code below:

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

It is not presently obvious that the algorithm is correct. So we will have a closer look at it:
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. As may be shown by induction, for every $0 \leq l \leq k + 1$, $s_{j+l+1} \geq p + \mathrm{length}(\zeta_1 x_1 ... \zeta_l x_l)$. 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 $p + d - 1$.
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 any knowledge of all the letters before s_j is of no use when "guessing" s_{j+1}).
Since $s_{k^*+3} - s_1 \geq s_{k+3} - s_1$ and $m - k \geq m - k^*$, 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 2.3. *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. Together with the fact that $X_1 = \mathfrak{M}[1] + 1$, we have

$$\Pr[X_1 = \log m + d + 1] < 2^{-d} \quad \text{for all integer } d \geq 0 \tag{2.1}$$

Note that all the X_j are stochastically independent and equally distributed (you do not gain any knowledge if you know how big the gap between the previous start positions was). So $\mathbf{E}[X_j] = \mathbf{E}[X_1]$ for all j. We will now show that $\mathbf{E}[X_j] < \log m + 3$:

$$\begin{aligned}
\mathbf{E}[X_j] = 1 + \mathbf{E}[\mathfrak{M}[1]] &= 1 + \log m + \mathbf{E}[\mathfrak{M}[1] - \log m] \\
&= 1 + \log m + \sum_{d=1}^{\infty} d \Pr[\mathfrak{M}[1] - \log m = d] \\
&= 1 + \log m + \sum_{d=1}^{\infty} d \Pr[X_1 = \log m + d + 1] \\
&< 1 + \log m + \sum_{d=1}^{\infty} d \left(\frac{1}{2}\right)^d \\
&= 1 + \log m + \frac{1}{2} / \left(1 - \frac{1}{2}\right)^2 = \log m + 3
\end{aligned}$$

Now let $Y_i := X_i - \frac{m-k^*}{k^*+2}$ and
apply Markov's inequality: $\Pr[X \geq h] \leq \mathbf{E}[X]/h$, for all $h > 0$ ($t > 0$):

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

Note that we used the independence and equal distribution of the variables Y_j, which follows from the independence and equal distribution of the variables X_j.
Inequality (2.1): $\Pr[X_1 = \log m + d + 1] < 2^{-d}$, 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 $t > 0$ ($\alpha := \log m + 1 - \frac{m-k^*}{k^*+2}$),

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

So we have up to now:

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

and choosing $t = \frac{\log_e 2}{2}$ and doing some algebra will yield:

$$
\begin{aligned}
&= (\sqrt{2}^{\alpha} \sum_{d=0}^{\infty} \sqrt{2}^{-d})^{k^*+2} \\
&\leq (\sqrt{2}^{\alpha+3.6})^{k^*+2} \\
&\leq \sqrt{2}^{(k^*+2)\log m-(m-k^*)+4.6(k^*+2)} \\
&< \sqrt{2}^{(5.6-c_1)(k^*+2)-(c_2-2)(c_1+\log m)}
\end{aligned}
$$

which is less than $1/m^3$ if $c_1 = 5.6$ and $c_2 = 8$.
So the probability to perform work at position s_1 and thus at each position is less than $1/m^3$. Thus 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

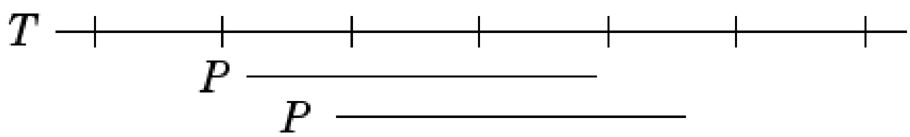
$$
\begin{aligned}
m^{-3}\mathcal{O}(k^2 \log m) &= m^{-3}\mathcal{O}(\frac{m^2}{(\log m)^2} \log m) \\
&= \mathcal{O}(\frac{1}{m \log m}) \\
&= \mathcal{O}(\lambda n.\lambda m.1)
\end{aligned}
$$

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

2.3.2 Sublinear Expected Time

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

The trick is to partition T into regions of length $\frac{m-k}{2}$. Then any substring of T that matches P must contain the whole of at least one region:



Now, starting from the left end of each region R, compute $k+1$ "maximum jumps" (using $\mathfrak{M}$, as in LV), 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:

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

□

To understand what an asset the algorithm is, consider the following facts:
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%.

2.4 Conclusion

> Gut gekaut ist halb verdaut.
>
> *German proverb*

The problem we have dwelt upon over the last few pages demonstrates in a beautiful manner how important a thorough preprocessing is in the business of algorithm design: Having gathered all the ingredients (data structures and auxiliary algorithms), merging them into the core algorithm was (although not obvious) quite short a task. – But that has been common knowledge for centuries: "A good chewing is half the digestion," as goes the translation of the above saying.