

Optimal Suffix Selection

Gianni Franceschini
Department of Computer Science
University of Pisa
francesc@di.unipi.it

S. Muthukrishnan
Google Inc.
New York
muthu@google.com

ABSTRACT

Given a string $S[1 \dots n]$, the *suffix selection* problem is to find the k th lexicographically smallest amongst the n suffixes $S[i \dots n]$, for $i = 1, \dots, n$. In particular, the fundamental question is if selection can be performed more efficiently than sorting all the suffixes.

If one considered n numbers, they can be sorted using $\Theta(n \log n)$ comparisons and the classical result from 70's is that selection can be done using $O(n)$ comparisons. Thus selection is provably more efficient than sorting, for n numbers.

Suffix sorting can be done using $\Theta(n \log n)$ comparisons, but does suffix selection need suffix sorting? We settle this fundamental problem by presenting an optimal, deterministic algorithm for suffix selection using $O(n)$ comparisons.

Categories and Subject Descriptors

F.2.2 [Analysis of Algorithms and Problem Complexity]: Nonnumerical Algorithms and Problems

General Terms

Algorithms, Theory

Keywords

Order statistics, selection, strings, suffixes

1. INTRODUCTION

We consider the comparison model where we count only the comparisons performed by an algorithm.¹ A collection of n numbers can be sorted using $\Theta(n \log n)$ comparisons. On the other hand, the famous five-author result [1] from early 70's shows that the problem of *selection* — choosing the number of *rank* k , that is, the k th smallest number —

¹Beyond the comparisons, all algorithms we consider in this paper take time linear in the total number of comparisons in the RAM model.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

STOC'07, June 11–13, 2007, San Diego, California, USA.
Copyright 2007 ACM 978-1-59593-631-8/07/0006 ...\$5.00.

can be solved using $O(n)$ comparisons in the worst case. Thus, selection can be done without sorting and is provably simpler than sorting in the comparison model. As a result, selection algorithms are quite useful in getting quick statistics about a collection of numbers: find extremes namely the smallest and the largest; find the median, that is, the number of rank $\lfloor n/2 \rfloor$, which is a more robust estimator than the average; filter the outliers by say, identifying the items defining the tenth and ninetieth percentiles, and retaining only the numbers that fall within this range; etc.

We study the sorting vs selection question for strings. Say $S = S[1 \dots n]$ is a string. The *suffix sorting* problem is to sort the suffixes $S[i \dots n]$, $i = 1, \dots, n$, in the *lexicographic* order. In the comparison model, we count the number of comparisons between the individual elements $S[i]$ vs $S[j]$ of the string that are performed by the algorithm for any i, j . If we treated each suffix as an “item” and applied the standard sorting algorithm, suffix sorting will take $\Theta(n^2 \log n)$ string element comparisons because “comparing” any two items will take $\Theta(n)$ string element comparisons in the worst case. Instead, after sorting the individual elements $S[1], \dots, S[n]$ using $O(n \log n)$ comparisons, we can reduce the problem to one over a string drawn from an alphabet of size at most n . Then, we can build what is known as the *suffix tree* [10, 9] of the string which takes $O(n)$ time [2]. The suffix tree directly gives the sorted order of the suffixes. Thus suffix sorting can be performed using $O(n \log n)$ comparisons; since the output of suffix sorting can be used to sort the individual elements $S[1], \dots, S[n]$, this is also the lower bound. There are a number of other methods to sort suffixes in $\Theta(n \log n)$ comparisons, e.g., as in [8]. Suffix sorting has many applications in pattern matching, compression, bioinformatics and information retrieval.

Curiously, we do not know the complexity of the associated *suffix selection* problem where the goal is to output the k th lexicographically smallest (equivalently, largest) suffix of S . Suffix selection is useful in analyzing the order statistics of suffixes in a string such as the extremes, medians and outliers, with potential applications in bioinformatics and information retrieval. A quick method for finding say the suffixes of rank $i(n/10)$ for each integer i , $0 \leq i \leq 10$, may be used to partition the space of suffixes for understanding the string better, load balancing and parallelization for truly massive strings such as DNA strings. Trivially, suffix selection can be solved by sorting all the suffixes as described above using $\Theta(n \log n)$ comparisons, but do we need to sort in order to select? This is a fundamental question, something to be discussed in textbooks.

To the best of our knowledge we are the first to address the problem of suffix selection. We present an *optimal, deterministic suffix selection algorithm that uses $O(n)$ comparisons*.

Often extending the complexity known for a collection of numbers to the collection of suffixes of a string is not straightforward. For example, sorting n integers in the range $[1, n]$ can be solved in $O(n)$ time by bucket sorting; sorting the suffixes of a string S drawn from such an integer alphabet $[1, n]$ in $O(n)$ time on the other hand was an open problem since 70's until it was settled in '96 [2]. If we only considered strings drawn from an integer alphabet $[1, n]$, then this suffix tree algorithm [2] will also solve the suffix selection problem in $O(n)$ time. However, our focus is on the comparison model where the elements of the string are accessible by comparisons only, as in standard sorting and order statistics problems.

Our algorithm for suffix selection uses the comparison-based five-author algorithm for selection [1]. It works by maintaining prefixes of a set of active suffixes at any moment, uses the five-author selection algorithm to narrow the active set and recurses by extending the prefixes of the suffixes that remain. Prefixes in the active set unfortunately intersect each other and naive algorithms will end up comparing each element in the string far too many times. We use *periodicity* properties of the substrings to jump over previously seen pieces of substrings and still use only $O(n)$ comparisons to solve the suffix selection problem. We also design and use various integer manipulation algorithms in the RAM model to make sure that in addition to the $O(n)$ comparisons, we use only $O(n)$ time computation.

2. OVERVIEW OF OUR ALGORITHMS

We let s_i denote the i th suffix of the input string S , i.e., $s_i = S[i] \cdots S[n]$.

Consider the following approach to the suffix selection problem.

The computation proceeds in phases. Each phase t is characterized by a prefix σ_t of the k th smallest suffix of S which is our target suffix, representing our current knowledge of the wanted suffix. Each phase t has a set of *active* suffixes of S containing all the suffixes with σ_t as a prefix, that is all the suffixes that could still be the k -th smallest suffix. Further, each phase has the number l_t of the suffixes of S less than any of the active suffixes in phase t . All the other suffixes are *inactive*. Our knowledge about the solution is augmented during the phase transitions. We start with the zeroth phase where σ_0 is void and any suffix is potentially our target; therefore, all suffixes are active. During the first phase transition we just apply the selection algorithm in [1] and find the k -th smallest element α_0 of S . The suffixes starting with α_0 are the active suffixes of the first phase, σ_1 contains just α_0 , and all the other suffixes become inactive. Now consider the transition from any phase t to phase $t + 1$. We focus on the (multi)set containing the $(t + 1)$ -th element of any active suffix and select from there the $(k - l_t)$ -th smallest element α_{t+1} . All the active suffixes with α_{t+1} as their $(t + 1)$ -th element are the active suffixes of phase $t + 1$ and we augment σ_t with α_{t+1} to obtain σ_{t+1} . Clearly α_{t+1} is the $(t + 1)$ -th element of the k -th smallest suffix of S and therefore our knowledge about the wanted suffix, i.e. σ_{t+1} , keeps growing. The computation ends when a phase transition leaves us with only one active suffix.

This solution needs $\Theta(n^2)$ comparisons in the worst case. However, we will follow this phase-based approach, but be more sophisticated about dealing with the *collisions of active suffixes* and using the work previously done on *inactive suffixes*, to get improved bounds.

How to Exploit Collisions of Active Suffixes.

For any phase t , the *extent* of a suffix s_i is the longest common prefix with σ_t . We say that two suffixes s_i, s_j *collide* when their extents are either adjacent (i.e. the last element of the extent of s_i is adjacent to the first element of the extent of s_j or vice versa) or overlapping.

Now let us consider the following modified solution to the suffix selection problem.

If no active suffixes collide, the computation proceeds just like in the previous solution. Let t' be the first phase in which two active suffixes collide. Since in any phase $t < t'$ the extents of the active suffixes have been expanded by one element at a time, the extents of any two active suffixes that collide in phase t' are adjacent rather than overlapping. For any active suffix s_i , the extent of s_i is followed by a certain number $r_i \geq 0$ of extents of other active suffixes and these extents are adjacent (not overlapping). The subsequence of extents associated with s_i is its *prospective extent*. Since the extent of an active element is $\sigma_{t'}$, the prospective extent of s_i is the periodic sequence $(\sigma_{t'})^{r_i}$. Any two active suffixes s_i and s_j with prospective extents $(\sigma_{t'})^{r_i}$ and $(\sigma_{t'})^{r_j}$ respectively with $r_i < r_j$ can be compared using only $O(1)$ element comparisons, provided that we know the lengths of the extents of all inactive suffixes. With the help of this property, we apply the algorithm in [1] directly to the active suffixes (instead of selecting the $(k - l_{t'})$ -th smallest element among the $(t' + 1)$ -th elements of the active suffixes, as we did in the previous solution). This produces a subset of active suffixes \mathcal{A}' containing the k -th smallest suffix such that if two active suffixes are in \mathcal{A}' then they have the same prospective extent and they also match at the elements right after their prospective extents (inverse of this claim does not necessarily hold). With this, the phase transition is concluded: \mathcal{A}' is the set of active suffixes of phase $(t' + 1)$ and $\sigma_{t'+1} = (\sigma_{t'})^r e'$ for some integer r and some element e' . It is easy to see that at most one active suffix per collision can still be active in phase $t' + 1$ and the extents of the active suffixes of phase $t' + 1$ do not overlap even though they can be adjacent and collide. The subsequent phase transitions proceed in the same way. By exploiting the collisions between active suffixes, this modified solution to the suffix selection problem can be shown to use $O(n \log n)$ comparisons. Still, this does not improve upon the complexity of suffix sorting for solving the suffix selection problem.

Reusing Work Done for Inactive Suffixes.

The two solutions presented thus far do not fully exploit the available information about inactive suffixes, i.e. their extents. The first solution does not even use the sizes of the extents of inactive suffixes while the second solution exploits this information only when the extents of inactive suffixes are used to compare two active suffixes. Ultimately, the issue to study is how to create the extent of an active suffix s_i in phase $t + 1$ during the transition from phase t . In the modified solution above, we add all the extents of the active suffixes that follow s_i and collide with it, and the element e' next to the right end of the extent of the last suffix in the

collision. This implies that the element e' can be accessed again $\omega(1)$ times in the subsequent phase transitions (e.g. s_i can later become inactive, e' can then be accessed again and added to the extent of another active suffix $s_{i'}$, $s_{i'}$ can then become inactive in turn, e' can later be accessed and added again and so forth). The challenge now is to avoid these multiple accesses. We say that a suffix is *maximal* if its extent is not contained in the extent of another suffix. Instead of adding only e' to the extent of s_i , our idea is to add the *whole extent of the maximal inactive suffix containing e'* .

As we will see, it is now difficult to blend together the ideas of exploiting collisions as well as reusing the work done for the inactive suffixes. One of the main obstacles in using both ideas simultaneously is that the collisions of active suffixes become less tractable because their extents as now defined may overlap in potentially arbitrary ways. The bulk of our technical contribution is in dealing with these obstacles to get a suffix selection algorithm that uses only $O(n)$ comparisons.

In what follows, we will first discuss the simple case of finding the largest (or smallest) suffix of S . Then we present our solution for the general suffix selection problem.

3. EXTREME SUFFIXES

We present an algorithm for selecting the lexicographically largest suffix of sequence $S[1 \dots n]$ using $O(n)$ comparisons; selecting the smallest is similar.

3.1 The Algorithm

The algorithm works in phases: the maximum suffix is gradually discovered through the phases.

3.1.1 Definitions and Notations

Any phase t is characterized by a *prefix* of the maximum suffix. This prefix, denoted by μ_t , represents the portion of the maximum suffix known at phase t . The *extent* of a suffix s_i , denoted by e_i , is the longest common prefix between s_i and μ_t ; if they do not have a common prefix, e_i is *degenerate* and is just $S[i]$. In any phase t , a suffix belongs to either the set \mathcal{A}_t of *active suffixes* or the set \mathcal{I}_t of *inactive suffixes*. A suffix s_i is *maximal* if there does not exist a suffix s_j such that $j < i$ and $|e_i| \leq |e_j|$. We denote the set of maximal suffixes by \mathcal{M}_t .

3.1.2 Initialization

We scan S and find the maximum element α_0 . Now \mathcal{A}_1 contains all suffixes s_i such that $S[i] = e\alpha_0$, \mathcal{I}_1 contains the remaining suffixes and $\mu_1 = \alpha_0$. The extent e_i of any suffix s_i is composed only of $S[i]$ and, if $s_i \in \mathcal{I}_1$, e_i is degenerate.

3.1.3 Phase Invariants

The following invariants hold for phase t .

- (i) μ_t is a prefix of the maximum suffix of S .
- (ii) \mathcal{A}_t contains s_i iff $e_i = \mu_t$.
- (iii) If $t > 1$, for any $s_i \in \mathcal{I}_t$ and $s_j \in \mathcal{A}_t$, we have that $|e_i| < |e_j|$.
- (iv) The extents of the maximal suffixes constitute a *non-overlapping covering* of S , that is $e_{i_1}e_{i_2} \dots e_{i_r} = S$, where $\{s_{i_h} | 1 \leq h \leq r\} = \mathcal{M}_t$ and $i_{h'} < i_{h''}$, for any $1 \leq h' < h'' \leq r$.

3.1.4 Phase Transition

Let us assume that the invariants hold after phase t ; we now describe phase $t+1$. Consider the non-overlapping covering of S induced by the extents of the maximal suffixes. This can be seen as a sequence of extents divided into subsequences $A_1 I_1 A_2 I_2 \dots A_l I_l$ of extents (A_1 and I_l may be empty) where for any $e_i \in A_h$ and $e_j \in I_h$ we have that $s_i \in \mathcal{A}_t$ and $s_j \in \mathcal{I}_t$, respectively. Intuitively, subsequences A_1, \dots, A_l represent the *collisions of active suffixes* we wrote about in § 2 and that we need to exploit. By the invariants maintained, we know that any active suffix is also maximal, and therefore part of the covering. For any $s_i \in \mathcal{A}_t$, let us consider the subsequence A_h such that $e_i \in A_h$ (that is, the collision that involves s_i). The *prospective extent* p_i of s_i is defined as the suffix starting with e_i of the subsequence $A_h e_h$, where e_h is the first extent in I_h if it exists. The phase transition proceeds as follows. Assume $|\mathcal{A}_t| > 1$, otherwise, we are done.

1. We find the prospective extent p_i of each $s_i \in \mathcal{A}_t$.
2. We find the lexicographically largest extent β_t and the set \mathcal{P}_t of suffixes which have that prospective extent. We set $\mathcal{A}_{t+1} = \mathcal{P}_t$, $\mathcal{I}_{t+1} = \mathcal{I}_t \cup (\mathcal{A}_t - \mathcal{P}_t)$ and $\mu_{t+1} = \beta_t$.
3. For any $s_i \in \mathcal{A}_t$, e_i is the longest common prefix with β_t .

3.1.5 Correctness and Complexity

We now sketch the correctness and complexity of the algorithm.

LEMMA 1. *Each phase transition maintains the invariants.*

LEMMA 2. *For any phase t , the transition from phase t to $t+1$ requires $O(|\mathcal{A}_t|)$ comparisons in the worst case.*

PROOF SKETCH. Step 1 requires a simple scan of set \mathcal{A}_t . Since prospective extents are defined only for active suffixes, the process in Step 2 involves $|\mathcal{A}_t|$ subsequences. We can show that (this is a key in the proof) $p_i = (e_i)^{r_i} e_{h_i} = (\mu_t)^{r_i} e_{h_i}$. The set $\{r_i | s_i \in \mathcal{A}_t\}$ can be trivially found and stored using $O(|\mathcal{A}_t|)$ comparisons and time. For any $s_i, s_j \in \mathcal{A}_t$, p_i and p_j can be compared using $O(1)$ element comparisons, given that both e_{h_i} and e_{h_j} are prefix of μ_t because $p_i = (\mu_t)^{r_i} e_{h_i}$, $p_j = (\mu_t)^{r_j} e_{h_j}$ and $r_i, r_j, |e_{h_i}|$ and $|e_{h_j}|$ can be retrieved in $O(1)$ time. Therefore, Step 2 takes $O(|\mathcal{A}_t|)$ comparisons in the worst case. Step 3 uses $O(|\mathcal{A}_t|)$ comparisons too for nearly the same reasons. \square

THEOREM 1. *The maximum suffix of a sequence S can be found using $O(n)$ comparisons in the worst case.*

PROOF SKETCH. The correctness of the algorithm follows directly from Lemma 1. The complexity of the algorithm is the sum of two quantities: the complexity of the initialization which is clearly $O(n)$, and the aggregate complexity of say p phase transitions in all, which by Lemma 2, is $O\left(\sum_{1 \leq t \leq p} |\mathcal{A}_t|\right)$. We will estimate the number of maximal suffixes of a generic phase t that cease to be maximal after the transition to phase $t+1$ is completed. Consider the non-overlapping covering of S induced by the extents of the maximal suffixes. That covering can be logically divided

into subsequences $A_1 I_1 A_2 I_2 \dots A_l I_l$ of extents where for any $e_i \in A_h$ and $e_j \in I_h$ we have $s_i \in \mathcal{A}_t$ and $e_j \in \mathcal{I}_t$, resp. (A_1 and I_l can be possibly empty but let us assume for simplicity that they are not.) Consider a subsequence $A_h I_h$ and let a_h be the number of extents in A_h . We can infer that maximal suffixes with extents in $A_h I_h$ that will not be maximal after transition are any $s_j \in \mathcal{M}_t$ such that $e_j \in A_h$ (except the suffix with the leftmost extent of A_h), and $s_{j'} \in \mathcal{M}_t$ such that $e_{j'} \in I_h$ and $e_{j'}$ is the leftmost extent in I_h . Hence, a_h of the maximal suffixes with extents in $A_h I_h$ will not be maximal anymore after the transition. Summing up, we have a total of $\sum_{1 \leq h \leq l} a_h = |\mathcal{A}_t|$ maximal suffixes of phase t that will cease to be so in phase $t+1$ (or $|\mathcal{A}_t| - 1$, if I_l is empty). When a suffix ceases to be maximal, it will never be maximal again. So the $O(|\mathcal{A}_t|)$ cost of the transition from phase t can be “charged” to the $\geq |\mathcal{A}_t| - 1$ maximal suffixes of phase t that will not be maximal anymore. So, the total number of comparisons in the phase transitions is $O(n)$. \square

4. GENERAL SELECTION

In this section we present an algorithm for selecting the k th lexicographically smallest suffix of $S[1 \dots n]$ using $O(n)$ comparisons.

4.1 The Algorithm

As for the special case of the largest suffix, the algorithm for selecting k th smallest suffix works in phases.

4.1.1 Definitions and Notations

As in previous section, any phase t of the general selection algorithm is characterized by a *prefix* of the suffix of rank k . This prefix, denoted by σ_t , represents the portion of the target suffix known at phase t . The *extent* of a suffix s_i , denoted by e_i , is either the longest common prefix between s_i and σ_t or just $S[i]$ in the *degenerate* case when they do not have a common prefix. In any phase t , a suffix is either in the set \mathcal{A}_t (it is *active*) or in \mathcal{I}_t (it is *inactive*). We denote with z_t the number of suffixes less than any of the active suffixes.

We need a few new definitions now. The *forward suffix* of a suffix s_i , denoted by f_i , is the leftmost suffix s_j among the ones in the set $\{s_{j'} \in \mathcal{I}_t \mid i < j' \leq i + |e_i|\}$ maximizing the quantity $j + |e_j| - 1$ (i.e. s_j is the leftmost suffix starting within the extent of s_i or right after it whose extent goes the farthest from the right end of e_i). Hence, only inactive suffixes can have forward suffixes. The *set of the entrant suffixes* of any suffix $s_i \in \mathcal{A}_t$, which we denote with $\mathcal{E}(i)$, is $\{s_j \in \mathcal{I}_t \mid i < j + |e_j| - 1 \leq i + |e_i| - 1 \text{ and } i < j\}$ (i.e. the inactive suffixes whose extents intersect e_i without being completely contained in it). Finally, we introduce an *overlapping cycle*. Let w be a sequence and u be a prefix of w . Let d be an integer d such that $0 \leq d \leq |u|$. Let $D = \langle d_1, \dots, d_r \rangle$ be a sequence of r integers such that $1 \leq d_i \leq |w|$. An *overlapping cycle with period w and tail u* , denoted with $[w, u](D, d)$, is the sequence $ww_{d_1}w_{d_2} \dots w_{d_{r-1}}w_{d_r}u_d$ such that w_{d_j} is the d_j -th suffix of w and u_d is the d -th suffix of u (if $d = 0$ the cycle has no tail).

4.1.2 Initialization

We apply linear time selection algorithm [1] to S and find the element α of rank k . Then \mathcal{A}_1 has each suffix s_i such that $S[i] = \alpha$, the remaining suffixes are in \mathcal{I}_1 and $\sigma_1 = \alpha$. The extent e_i of any suffix s_i is $S[i]$ and, if s_i is inactive, e_i

is degenerate. The forward suffix f_i of any $s_i \in \mathcal{A}_1 \cup \mathcal{I}_1$ is s_{i+1} . For any $s_i \in \mathcal{A}_1$, we set $\mathcal{E}(i) = \emptyset$. Finally, we set z_1 to be the number of elements of S less than α . Later, we will see in that we need other computations as part of the initialization too.

4.1.3 Phase Invariants

The following invariants hold for phase t .

- (I) σ_t is a prefix of the suffix of rank k of S .
- (II) \mathcal{A}_t contains s_i iff $e_i = \sigma_t$; z_t is the number of suffixes $s_i \in \mathcal{I}_t$ such that $s_i < s_j$, for any $s_j \in \mathcal{A}_t$.
- (III) If $t > 1$, for any $s_i \in \mathcal{I}_t$ and $s_j \in \mathcal{A}_t$, we have that $|e_i| < |e_j|$.
- (IV) For any two *consecutive* active suffixes $s_i, s_j \in \mathcal{A}_t$ (i.e. for any i' such that $i < i' < j$, $s_{i'} \in \mathcal{I}_t$), we have that $j \geq i + \lceil |e_i|/2 \rceil$ (i.e. the extents of s_i and s_j can overlap by less than a half of their lengths only)
- (V) If $t > 1$, there exist
 - (a) a sequence of integers $\langle d_1, \dots, d_r \rangle$ such that $1 \leq d_i \leq \lfloor |\sigma_{t-1}|/2 \rfloor$ and
 - (b) a forward suffix $f_i \in \mathcal{I}_{t-1}$ of $s_i \in \mathcal{A}_{t-1}$ and an integer d

such that $\sigma_t = [\sigma_{t-1}, s_j](\langle d_1, \dots, d_r \rangle, d)$ (that is, σ_t is an overlapping cycle with period σ_{t-1} and tail s_j but the overlapping of the “iterations” is limited).

- (VI) The forward suffix f_i of any $s_i \in \mathcal{A}_t \cup \mathcal{I}_t$ is known, i.e., the starting position of f_i is explicitly stored and retrievable in $O(1)$ time. The set of entrant suffixes $\mathcal{E}(i)$ of any $s_i \in \mathcal{A}_t$ is known and its members are retrievable in total $O(|\mathcal{E}(i)|)$ time.

4.1.4 Phase Transition: High Level Steps

Say the invariants hold for phase t . In this section we show a high level description of how to maintain the invariants while going to phase $t+1$ and while expanding our knowledge of the suffix of rank k (by invariants (I) and (V)). Later we will describe in detail how the crucial steps of the high level description are implemented.

We need some definitions. S can be logically divided into subsequences $O_1 I_1 O_2 I_2 \dots O_q I_q$ of elements with the following properties: (a) for any element $c \in O_p$ there exists a suffix $s_i \in \mathcal{A}_t$ such that $c \in e_i$; (b) for any element $c' \in I_p$ and for any suffix s_j such that $c' \in e_j$ we have that $s_j \in \mathcal{I}_t$. Subsequences O_1, \dots, O_l are *collisions of active suffixes*. We need to exploit them to achieve an optimal solution.

The *overlapping prospective extent* o_i of an active suffix s_i is defined as follows. Let O_p be the subsequence containing the extent of s_i . Let s_{i_p} be the rightmost active suffix starting within O_p . Let J_p be the prefix of subsequence I_p containing *only* elements belonging to the extent of the forward suffix f_{i_p} of s_{i_p} (J_p may be void). Then, o_i is the suffix of subsequence $O_p J_p$ starting from the leftmost element of s_i , i.e., the common subsequence between s_i and $O_p J_p$.

The total order defined for the overlapping prospective extents is a slight variation of the lexicographical order, the *overflow lexicographical order* or, briefly, *overflow order*. For any $o_{i'}$, the *overflow element* of $o_{i'}$ is $S[i' + |o_{i'}|]$. Note that

the overflow element of o_i is not part of o_i . Comparing o_i and o_j is defined as follows: if o_i is not a prefix of o_j and o_j is not a prefix of o_i , then o_i and o_j are compared lexicographically; if o_i is a prefix of o_j and $|o_i| < |o_j|$, then the overflow element of o_i and $S[j + |o_i|]$ are compared (the symmetrical case is analogous); if o_i and o_j match completely, then their overflow elements are compared. Two overlapping prospective extents o_i and o_j are *overflow-equal* if they match and their and their overflow elements are equal. The overflow equality and the overflow order are denoted by $\hat{=}$ and $\hat{<}$, respectively.

Let us present some basic properties of the overlapping prospective extents, without proofs.

For any $O_p I_p$ and for any $s_i \in \mathcal{A}_i$ starting within O_p , o_i of s_i is an *overlapping cycle with period σ_t and the extent of f_{i_p} as tail*, where s_{i_p} is the rightmost active suffix starting within O_p (as we anticipated in § 2, the extents of the active suffixes in a collision may overlap unpredictably). For any two suffixes $s_i, s_j \in \mathcal{A}_t$, if o_i is a prefix of o_j and $|o_i| < |o_j|$ then *the longest common prefix between s_i and s_j has length equal to $|o_i|$* . In other words, the $(|o_i| + 1)$ -th elements of s_i and s_j are different. For any collision O_p and for any $s_i, s_j \in \mathcal{A}_t$ starting within O_p , we have that $o_i \not\hat{=} o_j$.

The phase transition proceeds as follows.

1. If $|\mathcal{A}_t| = 1$, the suffix of rank k is the only active one left and the algorithm can terminate. Otherwise, we find the overlapping prospective extent o_i of each $s_i \in \mathcal{A}_t$.
2. We find the set \mathcal{O}_t containing the overlapping prospective extent o_{i_r} of rank $k - z_t$ (w.r.t. the overflow lexicographical order) and all the other o_j such that $o_{i_r} \hat{=} o_j$ (see § 4.2).

Let *the set of winners* \mathcal{W}_t be $\{s_i \in \mathcal{A}_t \mid o_i \in \mathcal{O}_t\}$. Let *the set of losers* \mathcal{L}_t be $\{s_i \in \mathcal{A}_t \mid o_i \notin \mathcal{O}_t\}$. Let b_t be the number of overlapping prospective extents $o_{j'}$ such that $o_{j'} \hat{<} o_{i_r}$.

3. We set $\mathcal{A}_{t+1} = \mathcal{W}_t$, $\mathcal{I}_{t+1} = \mathcal{I}_t \cup \mathcal{L}_t$, $z_{t+1} = z_t + b_t$ and we set $\sigma_{t+1} = o_j$, for any $o_j \in \mathcal{O}_t$ (they are all equal). For any $s_i \in \mathcal{W}_t$, we set $e_i = o_i$.
4. For any $s_i \in \mathcal{L}_t$, we set its extent e_i to be the longest common prefix with any $o_j \in \mathcal{O}_t$ (see § 4.2).
5. For any $s_i \in \mathcal{W}_t$, we update $\mathcal{E}(i)$ (see § 4.3).
6. For any $s_i \in \mathcal{L}_t$, we update the forward suffix of any entrant suffix $s_j \in \mathcal{E}(i)$ (see § 4.3).
7. For any $s_i \in \mathcal{W}_t$ or $\in \mathcal{L}_t$, we update its forward suffix f_i (see § 4.4).

LEMMA 3. *Each phase transition of the suffix selection algorithm maintains the invariants.*

4.2 Details: Overlapping Prospective Extents

Exploiting the structure in the problem, after suitable preprocessing, we show how to compare two overlapping prospective extents using $O(1)$ comparisons and show how to implement steps 2 and 4 of the phase transition. More details can be found in § A of the Appendix.

4.2.1 Preprocessing

Any overlapping prospective extent o_i of phase t can be seen as a sequence of partially overlapping occurrences of σ_t ended with a partially overlapping prefix of σ_t (the tail). The main part of o_t , formed by the overlapping occurrences of σ_t , is a suffix of a collision of active suffixes O_p . O_p is an overlapping cycle with period σ_t and no tail. In particular, by invariant (IV), for any O_p there exists a sequence of integers $G_p = \langle g_1, g_2, \dots, g_{r_p} \rangle$ such that $O_p = [\sigma_t, \langle \rangle] (G_p, 0)$, where $r_p + 1$ is the number of active suffixes starting within O_p and $1 \leq g_i \leq \lfloor |\sigma_t|/2 \rfloor$. Hence, if for any collision O_p we know the sequence of integers G_p “representing” it, then in order to compare two overlapping prospective extents o_i and o_j belonging to the collisions O_{p_i} and O_{p_j} , respectively, we can use suffixes of G_{p_i} and G_{p_j} instead of dealing with suffixes of o_i and o_j . More precisely, let us assume that o_i and o_j correspond to the x_i -th and x_j -th integers in G_{p_i} and G_{p_j} , respectively. Then, instead of finding the longest common prefix of o_i and o_j directly by comparing their elements, we can compare the $(x_i + 1)$ -th and $(x_j + 1)$ -th prefixes of G_{p_i} and G_{p_j} , respectively (we start from the $(x_i + 1)$ -th and $(x_j + 1)$ -th prefixes because the two integers $G_{p_i}[x_i]$ and $G_{p_j}[x_j]$ are calculated to describe the overlapping pattern of the whole collisions O_{p_i} and O_{p_j} and not of the two particular suffixes corresponding to o_i and o_j).

To exploit this idea we could proceed in the following way: *first*, we could concatenate the G_p 's into a single sequence $G = G_1 \langle 1 \rangle G_2 \langle 1 \rangle \dots \langle 1 \rangle G_q$ of $|\mathcal{A}_t|$ integers (the i -th interleaved sequence $\langle 1 \rangle$ corresponds to the leftmost active suffix starting within O_i); *second*, we could sort the suffixes of G ; *third*, we could process the sorted array of suffixes of G so that longest common prefix queries on the suffixes of G can be answered in $O(1)$ time. The preprocessing for phase t has to require $O(|\mathcal{A}_t|)$ time for the final algorithm to be linear. Unfortunately, the integers in G are suffix indexes of σ_t while $|G| = |\mathcal{A}_t|$ and tends to 1 over time. Therefore, the size of the alphabet of G is not linear in the cardinality of G and the sorting of the suffixes of G would require $\omega(|\mathcal{A}_t|)$ time.

The solution is to change the range of the integers in G from $[1 \dots |S|]$ to $[1 \dots |\mathcal{A}_t|]$. We use a table T of $|S|$ entries where each entry is a pair (timestamp, value). During initialization, we set up T so that $T[i] = (0, 0)$ for any $1 \leq i \leq |S|$. In phase t , after G is created, for each $j \in [1 \dots |S|]$: let $\langle t', v \rangle$ be the pair in $T[G[j]]$; if $t' = t$, we set $G[j] = v$; otherwise (i.e. $t' < t$ and the entry is old as it has been set in a previous phase t') we set $G[j] = j$ and $T[G[j]] = \langle t, j \rangle$. It is important to note that after the range reduction, for any two suffixes g_i, g_j of G , although the relative lexicographical order of g_i and g_j may not have been preserved, *their longest common prefix has the same length it had before*. After the range reduction we can carry out the preprocessing with a linear time suffix sorting algorithm for linear alphabets [7, 6, 4]) and the third step with prior work [5, 3]. Therefore:

LEMMA 4. *For any phase t , the preprocessing of the extent information requires $O(|\mathcal{A}_t|)$ comparisons and time linear in $O(|\mathcal{A}_t|)$.*

4.2.2 Comparing Overlapping Prospective Extents

The comparison between any two overlapping prospective extents o_x and o_y proceeds as follows. The *rightmost aligned suffixes* of o_x and o_y are the rightmost suffixes starting within o_x and o_y whose extents are aligned. Thanks to

the preprocessing, we can easily find the rightmost aligned suffixes $s_{x''}$ and $s_{y''}$ of o_x and o_y in $O(1)$ comparisons and time. (a) both $s_{x''}$ and $s_{y''}$ are not the rightmost active suffixes of o_x and o_y , respectively; (b) only one between $s_{x''}$ and $s_{y''}$ is the rightmost active suffixes of its overlapping prospective extent; (c) both $s_{x''}$ and $s_{y''}$ are the rightmost active suffixes of o_x and o_y . Given the periodicity properties of o_x and o_y , all the three cases are quite easily solved comparing just two aligned positions of the extents of two suffixes next two $s_{x''}$ and $s_{y''}$ (see § A.2 of the Appendix).

LEMMA 5. *For any phase t , after the preprocessing, any two overlapping prospective extents can be compared using $O(1)$ element comparisons.*

4.2.3 Summing Up

After the preprocessing, the set \mathcal{O}_t in step 2 of the phase transition can be found by applying the selection algorithm in [1] comparing any two o_i, o_j 's as described above. This comparison function can also find the length of their longest common prefix. Therefore,

LEMMA 6. *For any phase t , steps 2 and 4 of the phase transition from t to $t+1$ can be computed using $O(|\mathcal{A}_t|)$ comparisons and time.*

4.3 Details: Entrant Suffixes

Consider Step 5. We know that for any collision O_p and for any $s_i, s_j \in \mathcal{A}_t$ starting within O_p , we have that $o_i \not\equiv o_j$. Hence, we know that any two $o_i, o_j \in \mathcal{O}_t$ must belong to two different collisions O_{p_i} and O_{p_j} , respectively. Moreover, we know that the new inactive suffixes created during the phase transition from t to $t+1$ are the ones in $\mathcal{A}_t - \{s_i \in \mathcal{A}_t \mid o_i \in \mathcal{O}_t\}$. Clearly, for any s_i such that $o_i \in \mathcal{O}_t$, the only newly inactive suffixes in $\mathcal{I}_{t+1} \cap \mathcal{I}_t$ that can possibly enter in $\mathcal{E}(i)$ are the ones starting within O_{p_i} where O_{p_i} is the collision s_i is part of and hence we just need to examine them to verify if they have become entrant suffixes of s_i . Therefore:

LEMMA 7. *For any phase t , Step 5 of the phase transition uses $O(|\mathcal{A}_t|)$ comparisons and time.*

Consider Step 6. The actual computation of step 6 is simple. For any $s_i \in \mathcal{L}_t$ and for any $s_j \in \mathcal{E}(i)$, say the current forward suffix of s_j is s_{j_f} : if $i + |e_i| - 1 > j_f + |e_{j_f}| - 1$ then s_i is the new forward suffix of s_j and we update f_j . To analyze this algorithm, the key is what we show (without proof here) that *for any suffix $s_j \in \mathcal{I}_t$ there exists at most one $s_i \in \mathcal{A}_t$ such that $s_j \in \mathcal{E}(i)$* . Hence, we can conclude that any suffix $s_{j'}$, once it becomes inactive, can have its forward suffix updated only *once* during the entire execution of the suffix selection algorithm. Therefore:

LEMMA 8. *Over the entire execution of the suffix selection algorithm, Step 6 uses $O(|S|)$ comparisons and time.*

4.4 Details: Updating Forward Suffixes

We briefly describe two data structures and use them to implement step 7.

4.4.1 Structure I

Our structure is defined on *pairs* of small integers $\langle p, v \rangle$ composed of a *position* p and a *value* v , each of $\log \ell^c$ bits,

where c is a constant (we will constrain ℓ later). Pairs form b lines L_0, L_1, \dots, L_b , for a constant $b \leq c$. For any i , line L_i is a sequence of $|S|/\ell^i$ pairs. Any line L_i is logically divided into *groups* $G_1^i, G_2^i, G_3^i, \dots$ each one containing ℓ consecutive pairs of small integers. There is a hierarchy among the lines. The lowest one in the hierarchy is L_0 , the longest one. Any group $G_j^i \in L_i$ is associated with a subsequence of contiguous groups in L_{i-1} . More specifically, for any $i > 0$, G_j^i of L_i is associated with the sequence of ℓ contiguous groups $G_{(j-1)\ell+1}^{i-1}, G_{(j-1)\ell+2}^{i-1}, \dots, G_{j\ell}^{i-1}$ of L_{i-1} . Given this association between adjacent levels, it is easy to see the structure as a forest of complete ℓ -ary trees where any group is one node, the ℓ groups in the immediately lower line associated with the group are its children and all the roots of the trees reside in the highest line. For any group $G \in L_i$, we denote with $par(G)$ its parent group and with $ch_h(G)$ its h -th children, i.e., $G_{(j-1)\ell+h}^{i-1}$, if G is the j -th group of L_i . Clearly the starting positions of $par(G)$ and any $ch_h(G)$ in their respective lines L_{i+1} and L_{i-1} can be calculated in $O(1)$ time (we choose ℓ as a power of 2). For any group $G \in L_i$ we denote with $L_o(G)$ the subsequence of L_0 composed by the groups that are descendants of G and we denote with $l_{0,i}$ the size of it, that is the number of pairs in $L_o(G)$ (it is the same for any group in the i -th line). So we have that $l_{0,i} = \ell^{i+1}$, for $i \geq 0$, and let us assume by convention that $l_{0,-1} = 0$ (to spare a special case in the invariants and tables definitions). We refer to the values of the pairs in L_0 as the *originals* because, as we will see, any value v_j of a pair in a line L_i , $i > 1$, is a copy of a value in L_0 . If \mathcal{A} is a multiset (and hence multiple maxima may coexist in \mathcal{A}) then we set $\arg \max_y \mathcal{A}$ to be the *smallest y corresponding to a maximum*. We have the following invariants.

- (A) For any $G \in L_0$, for the j -th pair $\langle p_j, v_j \rangle$ of G we have that $p_j = j$.
- (B) For any $i > 0$ and for any $G \in L_i$, for the j -th pair $\langle p_j, v_j \rangle$ of G we have that:
 - v_j is equal to v_x of the x -th pair $\langle p_x, v_x \rangle$ of $ch_j(G)$ such that $x = \arg \max_y \left\{ (y-1)l_{0,i-2} + p_y + v_y - 1 \mid \langle p_y, v_y \rangle \text{ is the } y\text{-th pair of } ch_j(G) \right\}$.
 - p_j is equal to the position in $L_o(ch_j(G))$ of the v_j 's original (the position is relative to the subsequence $L_o(ch_j(G))$, not the whole L_0).

We also use b lookup tables F_0, F_1, \dots, F_b . Table F_i corresponds to line L_i and represents the lookup implementation of the function $\phi_i : \mathbb{N}^{2\ell} \rightarrow \mathbb{N}$, $i \geq 0$, such that, for any sequence of ℓ pairs of small integers $\langle \langle p_1, v_1 \rangle, \dots, \langle p_\ell, v_\ell \rangle \rangle$, $\phi_i(\langle \langle p_1, v_1 \rangle, \dots, \langle p_\ell, v_\ell \rangle \rangle) = \arg \max_{1 \leq j \leq \ell} \left\{ (j-1)l_{0,i-1} + p_j + v_j - 1 \mid \langle p_j, v_j \rangle \text{ s.t. } v_j \neq 0 \right\}$.

We use these structures to support the following query:

DEFINITION 1. *Given an interval $[q \dots q']$, $1 \leq q \leq q' \leq |S|$, such that $q' - q + 1 \leq \ell^{b+1}$, we want to find the pair $\langle p_x, v_x \rangle \in L_0$ such that $x = \arg \max_y \left\{ y + v_y - 1 \mid q \leq y \leq q' \right\}$, where $\langle p_y, v_y \rangle$ is the y -th pair of L_0 .*

This query is answered as follows. We find the lowest line L_i such that $l_{0,i} \geq q' - q + 1$. We find the (at most) two

consecutive groups $G, G' \in L_i$ such that the concatenation of $L_o(G)$ and $L_o(G')$ completely contains the subsequence $L_0[q \dots q']$. Let us suppose that there are in fact two of them (the other case is analogous) and let q'' be the integer such that $L_0[q \dots q'' - 1] \subseteq L_o(G)$ and $L_0[q'' \dots q'] \subseteq L_o(G')$. We find the smallest h of G' such that $L_0[q'' \dots q']$ is contained in the concatenation of $L_o(ch_1(G')), \dots, L_o(ch_h(G'))$. Let q''' be the index (in L_0) of the leftmost pair in $L_o(ch_h(G'))$. We solve the query for the interval $[q''' \dots q']$ recursively. Let $\langle p_z, v_z \rangle \in L_0$ be the pair satisfying the sub-query. We extract from G' the subsequence $G'[1 \dots h - 1]$. By careful memory layout of the lines (see § B.1 of the Appendix), this result will already be a full-sized integer j that can be used to index F_i . Using $F_i[j]$, we find the pair $\langle p_y, v_y \rangle \in L_0$ satisfying the query for the interval $[q'' \dots q''' - 1]$. We repeat for $L_0[q \dots q'' - 1]$ what we have done for $L_0[q'' \dots q']$ thereby finding the pairs $\langle p_{z'}, v_{z'} \rangle, \langle p_{y'}, v_{y'} \rangle \in L_0$ satisfying the (at most) two corresponding sub-queries. We return $\arg \max_{j \in \{z', y', y, z\}} \{j + v_j - 1\}$.

We omit the details of how this structure can be maintained under updates to L_0 , how the lookup table for F_i 's is built, and how the lines are laid out in memory for the desired access. We choose ℓ properly so that using the power of bit operations on small integers in the RAM model (in particular, shift left or right by one position), we can implement these operations efficiently. As sketched here, we actually need shift by an arbitrary number of positions, but a slightly more sophisticated version can do with only left or right shift by one position in unit time. Some details can be found in § B.1 of the Appendix.

LEMMA 9. *Structure I can be built using $O(|S|)$ time and comparisons and updated, queried in $O(1)$ time.*

4.4.2 Structure II

Suppose that $|S|$ is a power of 2. The structure is a complete binary tree \mathcal{T} with $|S|$ leaves. For any node u of \mathcal{T} , the *base tree of u* is the complete subtree of \mathcal{T} rooted in u and containing all the leaves of \mathcal{T} that are descendants of u . The i -th leaf l_i is associated with the i -th suffix s_i of S and contains an integer value v_i . Any internal node u of \mathcal{T} contains a pair $\langle p, v \rangle$: v is equal to the largest value contained in the leaves of the base tree \mathcal{T}_u of u ; p is a pointer to the leftmost leaf of \mathcal{T}_u having its integer value equal to v . For any query interval $[x \dots y]$, $1 \leq x \leq y \leq |S|$, the leftmost leaf l_i with the largest value v_i among the leaves in $\{l_j | x \leq j \leq y\}$ can be found as follows. The *tree-cover* for the sequence of leaves $l_x l_{x+1} \dots l_y$ is a set $\{\mathcal{T}_1, \mathcal{T}_2, \dots, \mathcal{T}_r\}$ of subtrees of \mathcal{T} such that (a) any \mathcal{T}_p is a base tree; (b) any two $\mathcal{T}_p, \mathcal{T}_{p'}$ are disjoint; (c) $l_x l_{x+1} \dots l_y = \mathcal{L}(\mathcal{T}_1) \mathcal{L}(\mathcal{T}_2) \dots \mathcal{L}(\mathcal{T}_r)$, where $\mathcal{L}(\mathcal{T}_p)$ is the sequence of leaves of \mathcal{T}_p .

1. We find a tree-cover \mathcal{M} for $l_x l_{x+1} \dots l_y$ as follows. Let $u = l_x$ and $\mathcal{M} = \emptyset$.
 - (a) If u is the right child of $\text{par}(u)$ (the parent of u), let l_j be the rightmost leaf of the base tree of u . We set $\mathcal{M} = \mathcal{M} \cup \{u\}$, $u = l_{j+1}$ (i.e. the leaf next to l_j) and we proceed with step 1d.
 - (b) Otherwise, let l_j be the rightmost leaf of the base tree of $\text{par}(u)$. If $j > y$ (that is l_j is out of the query interval), let $l_{j'}$ be the rightmost leaf of the base tree of u . We set $\mathcal{M} = \mathcal{M} \cup \{u\}$, $u = l_{j'+1}$ and we proceed with step 1d.

- (c) Otherwise, we set $u = \text{par}(u)$ and we start again with step 1a.
- (d) If all the leaves in $l_x l_{x+1} \dots l_y$ are covered (i.e. if l_y has been accessed during the previous steps) then we proceed with step 2. Otherwise, we start again with step 1a.

2. We find the leftmost tree among the ones in \mathcal{M} whose root has a pair $\langle p, v \rangle$ with the largest v (since they are disjoint base trees, their order derives from the one of the leaves). We return the leaf pointed to by p .

Note that the tree-cover \mathcal{M} found in step 1 is *minimal*, that is for any other tree-cover \mathcal{M}' for $l_x l_{x+1} \dots l_y$, we have that $|\mathcal{M}| < |\mathcal{M}'|$. It is easy to see that a minimal tree cover cannot contain more than $2 \log |S|$ subtrees. Therefore, we can show:

LEMMA 10. *Structure II used can be built in $O(|S|)$ time, updated in $O(\log |S|)$ time and queried in $O(\log^2 |S|)$ time.*

4.4.3 Summing Up

Step 7 is easy for $s_i \in \mathcal{W}_t$. For any suffix $s_i \in \mathcal{A}_t$ in this step, $o_i \in \mathcal{O}_t$ and s_i 's new extent is o_t . Let $s_{i'}$ be the rightmost active suffix starting within o_t , we set f_i to be the forward suffix of $f_{i'}$ and we are done. Consider any $s_i \in \mathcal{L}_t$. In addition to the initialization discussed in other steps thus far, we preprocess for Structure I with following rule: if the j -th suffix s_j of S is in \mathcal{I}_1 , the value v_j of the j -th pair in L_0 and all its copies in the higher lines is set to $|e_j|$; otherwise if $s_j \in \mathcal{A}_1$, v_j is set to $-\infty$ (note that the active suffixes cannot be forward suffixes). We now have two kinds of phases: the *early* and *late* phases such that $\sigma_t \leq \alpha \log^2 |S|$ and $\sigma_t > \alpha \log^2 |S|$.

Early phases. For any $s_i \in \mathcal{L}_t$, we change the value v_i of the i -th pair in L_0 from $-\infty$ to e_i (s_i was active in phase t and will be inactive in $t+1$) and we update Structure I. Then, for any $s_i \in \mathcal{L}_t$, we query Structure I with the interval $[i+1 \dots i+|e_i|]$ and obtain the (possibly new) forward suffix of s_i .

Interlude. The interlude is the phase transition from a phase t' to a phase t'' such that $\sigma_{t'} \leq \alpha \log^2 |S|$ and $\sigma_{t''} > \alpha \log^2 |S|$. We proceed as before but after setting forward suffixes for $s_i \in \mathcal{W}_t$, we build Structure II. The i -th leaf l_i is associated with the i -th suffix s_i of S and contains an integer value v_i set as follows: if $s_i \in \mathcal{A}_{t''}$ then $v_i = -\infty$; otherwise, if $s_i \in \mathcal{I}_{t''}$ then $v_i = i + |e_i| - 1$. Then while dealing with $s_i \in \mathcal{L}_t$, we use the newly built Structure II.

Late phases. For any $s_i \in \mathcal{L}_t$, we change the value v_i of the i -th leaf of T from $-\infty$ to e_i and update Structure II. Then, for any $s_i \in \mathcal{L}_t$, we query Structure II with the interval $[i+1 \dots i+|e_i|]$ and obtain the (possibly new) forward suffix of s_i .

In the early phases, we execute one update and one query of Structure I for any suffix in $\bigcup_{t < t'} \mathcal{L}_t$, where t' is the last early phase and $|\bigcup_{t < t'} \mathcal{L}_t| = O(|S|)$. In the interlude and the late phase period we build Structure II and do one update and one query for any suffix in $\bigcup_{t \geq t'} \mathcal{L}_t$. We know that for any late phase t , $\sigma_t > \alpha \log^2 |S|$. Because of invariants (IV) and (II), we know that $|\bigcup_{t \geq t'} \mathcal{L}_t| = O(|S| / |\sigma_{t'}|) = O(|S| / \log^2 |S|)$. Hence,

LEMMA 11. *During the entire execution of the suffix selection algorithm, the total time and comparisons needed to perform Step 7 in the phase transitions is $O(|S|)$.*

4.5 Correctness and Complexity

Combining all the pieces gives us the main result:

THEOREM 2. *The lexicographically k -th smallest suffix of a sequence $S[1 \dots n]$ can be deterministically found using $O(|S|)$ comparisons and time in the worst case.*

5. REFERENCES

- [1] M. Blum, R. Floyd, V. Pratt, R. Rivest, and R. Tarjan. Time bounds for selection. *J. Comput. System Sci.*, 7:448–61, 1973.
- [2] M. Farach. Optimal suffix tree construction with large alphabets. In *Proceedings of the 38th Annual Symposium on Foundations of Computer Science (FOCS)*, pages 137–143. IEEE Computer Society Press, 1997.
- [3] D. Harel and R. Tarjan. Fast algorithms for finding nearest common ancestors. *SIAM Journal on Computing*, 13(2):338–355, May 1984.
- [4] J. Karkkainen and P. Sanders. Simple linear work suffix array construction. In *Proc. Int. Colloquium on Automata, Languages and Programming*, volume 2719 of *Lecture Notes in Computer Science*, pages 943–955, 2003.
- [5] T. Kasai, G. Lee, H. Arimura, S. Arikawa, and K. Park. Linear-time longest-common-prefix computation in suffix arrays and its applications. In *Proc. Annual Symposium on Combinatorial Pattern Matching*, volume 2089 of *Lecture Notes in Computer Science*, pages 181–192, 2001.
- [6] D. K. Kim, J. S. Sim, H. Park, and K. Park. Linear-time construction of suffix arrays. In *Proc. Annual Symp. on Combinatorial Pattern Matching*, volume 2676 of *Lecture Notes in Computer Science*, pages 186–199, 2003.
- [7] P. Ko and S. Aluru. Space efficient linear time construction of suffix arrays. In *Proc. Annual Symposium on Combinatorial Pattern Matching*, volume 2676 of *Lecture Notes in Computer Science*, pages 200–210, 2003.
- [8] U. Manber and G. Myers. Suffix arrays: A new method for on-line string searches. *SIAM Journal on Computing*, 22(5):935–948, 1993.
- [9] E. M. McCreight. A space-economical suffix tree construction algorithm. *Journal of the ACM*, 23(2):262–272, 1976.
- [10] P. Weiner. Linear pattern matching algorithms. In *IEEE Symposium on Foundations of Computer Science (FOCS)*, 1973.

APPENDIX

A. DEALING WITH THE OVERLAPPING PROSPECTIVE EXTENTS

A.1 The Preprocessing

We have four main steps.

1. By the definition of overlapping prospective extent, every subsequence O_p is an overlapping cycle with period σ_t and no tail. In particular, by invariant (IV), for any O_p there exists a sequence of integers $G_p = \langle g_1, g_2, \dots, g_{r_p} \rangle$ such that $O_p = [\sigma_t, \langle \rangle](G_p, 0)$, where $r_p + 1$ is the number of active suffixes starting within O_p and $1 \leq g_i \leq \lfloor |\sigma_t|/2 \rfloor$. We compute the sequence of integers G_p for any O_p .
2. We build the following three structures.
 - (a) Let G'_p be $\langle 1 \rangle G_p$ (i.e. the concatenation of the sequence containing only 1 with G_p). We concatenate all the G'_p 's into a single sequence $G = G'_1 G'_2 \dots G'_q$ of $|\mathcal{A}_t|$ integers. The i -th interleaved sequence $\langle 1 \rangle$ corresponds to the leftmost active suffix starting within O_i .
 - (b) We compute (and store as a table) the function $\tau : \{1, \dots, |\mathcal{A}_t|\} \rightarrow \{1, \dots, |S|\}$ such that $s_{\tau(i)}$ is the suffix whose extent correspond to the integer $G[i]$.
 - (c) We compute (and store as a table) the function $\lambda : \{1, \dots, |\mathcal{A}_t|\} \rightarrow \{1, \dots, |\mathcal{A}_t|\}$ such that $\lambda(i)$ is the position in G of the rightmost integer in G'_p , where G'_p is the original sequence $G[i]$ came from (see definition of G in 2a),
3. We need to change the range of the integers in G from $[1 \dots |S|]$ to $[1 \dots |\mathcal{A}_t|]$. We use a table T of $|S|$ entries where each entry is a pair $\langle \text{timestamp}, \text{value} \rangle$. Let us suppose that during the initialization of the algorithm we also set up T so that $T[i] = \langle 0, 0 \rangle$ for any $1 \leq i \leq |S|$. The reduction consists of the following simple process. For each $j \in [1 \dots |S|]$: let $\langle t', v \rangle$ be the pair in $T[G[j]]$; if $t' = t$, we set $G[j] = v$; otherwise (i.e. $t' < t$ and the entry is old as it has been set in a previous phase t') we set $G[j] = j$ and $T[G[j]] = \langle t, j \rangle$.
4. After the step 3 we have that each one of the $|\mathcal{A}_t|$ entries of G belongs to $\{1, \dots, |\mathcal{A}_t|\}$. In this step we first sort the suffixes of G using a linear time suffix sorting algorithm for linear alphabets (e.g. [7, 6, 4]). Then, we process the suffix array of G so that longest common prefix queries on the suffixes of G can be answered in $O(1)$ time (e.g. using [5] and [3]).

A.2 Comparing Two Overlapping Prospective Extents Efficiently

Let us recall that by the definition of overlapping prospective extent, any overlapping prospective extent o_x is composed by a number of extents of active suffixes and the extent of a forward suffix. The *rightmost aligned suffixes* of any two overlapping prospective extents o_x and o_y are the rightmost suffixes starting within o_x and o_y whose extents are aligned.

The comparison between o_x and o_y proceeds with two steps.

1. In this step we find the rightmost aligned suffixes $s_{x''}$ and $s_{y''}$ of o_x and o_y . Let $s_{x'} \in \mathcal{A}_t$ be the suffix whose extent $e_{x'}$ is the second one (from the left end) of o_x and let $i_{x'}$ be the index of $s_{x'}$ in a left-to-right enumeration of \mathcal{A}_t (i.e. $s_{x'}$ is the $i_{x'}$ -th active suffix starting from the left end of S). Let us assume similar definitions for o_y . We proceed with three substeps.

- (a) Using the structures from the preprocessing, we find the length of the longest common prefix between $G_{i_{x'}}$ (the $i_{x'}$ -th suffix of G) and $G_{i_{y'}}$. Let it be l .
 - (b) If $i_{x'} + l - 1 > \lambda(i_{x'})$ or $i_{y'} + l - 1 > \lambda(i_{y'})$, then let $l = \min(\lambda(i_{x'}) - i_{x'} + 1, \lambda(i_{y'}) - i_{y'} + 1)$.
 - (c) Let $x'' = \tau(i_{x'} + l - 1)$ and $y'' = \tau(i_{y'} + l - 1)$.
2. We are finally able to decide the relative order of o_x and o_y . We have to distinguish three cases.

- (a) Let us assume that both $s_{x''}$ and $s_{y''}$ are not the rightmost active suffixes of o_x and o_y . Let $s_{x''''}$ and $s_{y''''}$ be the active suffixes next to $s_{x''}$ and $s_{y''}$, respectively. Finally, let us assume that $x''' > y'''$ (the other case is symmetric). Let us consider suffix s_{x_y} , where $x_y = x'' + y''' - y''$ (that is the suffix starting within o_x that is aligned to $s_{y''''}$ of o_x). Since $s_{x''''}$ is the active suffix next to $s_{x''}$, we have that $s_{x_y} \in \mathcal{I}_t$ and hence $|e_{x_y}| < |e_{y''''}|$. Therefore, the relative order of o_x and o_y can be decided in the following ways:

- i. if e_{x_y} is degenerate, by comparing the elements $S[x_y]$ and $S[y''' + |e_{x_y}|]$;
- ii. if e_{x_y} is not degenerate, by comparing $S[x_y + |e_{x_y}|]$ and $S[y''' + |e_{x_y}|]$.

- (b) Let us assume that only $s_{x''}$ is the rightmost active suffixes of its overlapping prospective extent o_x . Let $s_{x''''}$ be the forward suffix $f_{x''}$ of $s_{x''}$. Let $s_{y''''}$ be the active suffixes next to $s_{y''}$ in o_y . By the assumption on $s_{x''}$, we know that $s_{x''''} \in \mathcal{I}_t$ and hence $|e_{x''''}| < |e_{y''''}|$. Therefore, the relative order of o_x and o_y can be decided in the following ways:

- i. if $e_{x''''}$ is degenerate, by comparing the elements $S[x'''']$ and $S[y''' + |e_{x''''}|]$;
- ii. if $e_{x''''}$ is not degenerate, by comparing $S[x'''' + |e_{x''''}|]$ and $S[y''' + |e_{x''''}|]$.

- (c) Let us assume that both $s_{x''}$ and $s_{y''}$ are the rightmost active suffixes of o_x and o_y . Let $s_{x''''}$ and $s_{y''''}$ be the forward suffixes $f_{x''}$ and $f_{y''}$ of $s_{x''}$ and $s_{y''}$, respectively. We have the following cases.

- i. If at least one of the extents of $s_{x''''}$ and $s_{y''''}$ is degenerate, the relative order of o_x and o_y is decided by comparing the elements $S[x + |o_x|]$ and $S[y + |o_y|]$ (in this case o_x and o_y have the same length).
- ii. Otherwise, let us assume that $|o_x| \geq |o_y|$. Let s_{y_x} be the suffix within o_y that is aligned to $s_{x''''}$ (that is $y_x = y''' + x'''' - x$). The relative order of o_x and o_y is decided by comparing the elements $S[s_{y_x} + |e_{y_x}|]$ and $S[s_{x''''} + |e_{y_x}|]$.

B. UPDATING THE FORWARD SUFFIXES OF WINNERS AND LOSERS

B.1 Additional Details about Structure I

B.1.1 The Memory Layout

We assume the availability of the shift as a basic, unit cost operator (that is, an integer can be multiplied and divided

by 2^i in constant time). This assumption is not strictly necessary since we can describe a slightly more complex structure that makes use of multiplications and divisions by 2 only. The memory layout of the lines exploits the availability of the shift operator. In the comparison model/RAM, besides the completely abstract input elements, it is possible to employ integer values that can be stored in single locations of memory and that can be directly used with the allowed arithmetic operators. These full-size integers are presumed to have at least $\log |S|$ bits. We *do not* store each small integer as a full-size integer. Instead we store *each group* G as a single full-size integer (as a we will see later, our choice for ℓ guarantee that $2\ell \log \ell^c < \log |S|$). Hence, any line L_i is completely contained in $\frac{|S|}{\ell^i \cdot \ell \log \ell^c}$ locations of memory ($|S|/\ell^i$ is the number of pair of small integers in L_i). It is easy to see that any contiguous subsequence of L_i with $\ell' \leq \ell$ pairs of small integers can be copied from L_i using $O(1)$ shifts.

B.1.2 Building the Lookup Tables

The b lookup tables are built in the following way. Using the shift operator, any sequence $\langle \langle p_1, v_1 \rangle, \dots, \langle p_\ell, v_\ell \rangle \rangle$ of pairs of small integers can be easily compacted into one full-size integer j in $O(\ell)$ time. $F_i[j]$ will contain the integer resulting from the computation of $\phi_i(\langle \langle p_1, v_1 \rangle, \dots, \langle p_\ell, v_\ell \rangle \rangle)$. Since our target time complexity is linear in $|S|$, we have to choose ℓ so that the following constraint is respected:

$$C_\phi(\ell) \cdot 2^{2\ell \log \ell^c} = O(|S|),$$

where $C_\phi(\ell)$ is the worst case complexity of applying any ϕ_i to a sequence of ℓ pairs (the total number of entries of F_i is $2^{2\ell \log \ell^c}$, given that a sequence contains ℓ pairs of integers of $\log \ell^c$ bits each). Since $C_\phi(\ell)$ is clearly $\Theta(\ell)$, we can easily choose ℓ so that the following conditions hold:

- (i) $\ell = 2^z$ for some integer z ,
- (ii) $\ell = \Theta(\sqrt{\log |S|})$.

B.1.3 Queries and Updates

The queries in Definition 1 can be answered in seven steps:

1. We find the lowest line L_i such that $l_{0,i} \geq q' - q + 1$.
2. We find the (at most) two consecutive groups $G, G' \in L_i$ such that the concatenation of $L_o(G)$ and $L_o(G')$ completely contains the subsequence $L_0[q \dots q']$. Let us suppose that there are in fact two of them (the other case is analogous) and let q'' be the integer such that $L_0[q \dots q'' - 1] \subseteq L_o(G)$ and $L_0[q'' \dots q'] \subseteq L_o(G')$.
3. Then, we find the smallest integer h of G' such that $L_0[q'' \dots q']$ is completely contained in the concatenation of $L_o(ch_1(G')), L_o(ch_2(G')), \dots, L_o(ch_h(G'))$. Let q''' be the index (in L_0) of the leftmost pair in $L_o(ch_h(G'))$.
4. We solve the query for the interval $[q'' \dots q']$ recursively. Let $\langle p_z, v_z \rangle \in L_0$ be the pair satisfying the subquery.
5. Using the shift operator, we extract from G' the subsequence $G'[1 \dots h - 1]$. By the memory layout of the lines, the result is already in in form of a full-size integer j that can be used to index F_i . Using $F_i[j]$, we immediately find the pair $\langle p_y, v_y \rangle \in L_0$ satisfying the query for the interval $[q'' \dots q''' - 1]$.

6. We repeat for $L_0[q \dots q'' - 1]$ what we have done for $L_0[q'' \dots q']$ in steps 3, 4 and 5, thereby finding the pairs $\langle p_{z'}, v_{z'} \rangle, \langle p_{y'}, v_{y'} \rangle \in L_0$ satisfying the (at most) two corresponding sub-queries.

7. We return $\arg \max_{j \in \{z', y', y, z\}} \{j + v_j - 1\}$.

Finally, maintaining the invariants of the structure whenever a value in L_0 is updated is quite straightforward. Let us suppose that the value v of a pair of the j -th group $G \in L_0$ is changed. We proceed as follows.

1. We apply ϕ_0 to G (that is, by the memory layout for the lines, we index F_0 with the integer in the j -th location of L_0) and we find (a reference to) a pair $\langle p_i, v_i \rangle \in G$.
2. Let G be the j -th child of $\text{par}(G)$, if the pair referenced by the position p_j stored in the pair $\langle p_j, v_j \rangle \in \text{par}(G)$ is not $\langle p_i, v_i \rangle \in G$, we update $\langle p_j, v_j \rangle$ accordingly.
3. We apply recursively the first two steps to $\text{par}(G) \in L_1$ (obviously, using ϕ_1 and F_1 instead of ϕ_0 and F_0).