Fast Order-Preserving Pattern Matching

Sukhyeun Cho[1], Joong Chae Na[2], Kunsoo Park[3], and Jeong Seop Sim[1,⋆]

[1] School of Computer and Information Engineering, Inha University, Korea
csukhyeun@inha.edu, jssim@inha.ac.kr
[2] Department of Computer Science and Engineering, Sejong University, Korea
jcna@sejong.ac.kr
[3] School of Computer Science and Engineering, Seoul National University, Korea
kpark@theory.snu.ac.kr

Abstract. Given a text T and a pattern P, the order-preserving pattern matching (OPPM) problem is to find all substrings in T which have the same relative orders as P. The OPPM has been studied in the fields of finding some patterns affected by relative orders, not by their absolute values. For example, it can be applied to time series analysis like share prices on stock markets and to musical melody matching of two musical scores. In this paper, we present a new method of deciding the order-isomorphism between two strings even when there are same characters. Then, we show that the bad character rule of the Horspool algorithm for generic pattern matching problems can be applied to the OPPM problem. Finally, we present a fast algorithm for the OPPM problem and give experimental results to show that our algorithm is about 2 to 5 times faster than the KMP-based algorithm in reasonable cases.

Keywords: order-preserving pattern matching, order-isomorphism, Horspool algorithm.

1 Introduction

Given a text T and a pattern P, the order-preserving pattern matching (OPPM for short) problem is to find all substrings in T which have the same relative orders as P. For example, when $P = (35, 40, 23, 40, 40, 28, 30)$ and $T = (10, 20, 15, 28, 32, 12, 32, 32, 20, 25, 15, 25)$ are given, P has the same relative orders as the substring $T' = (28, 32, 12, 32, 32, 20, 25)$ of T. In T' (resp. P), the first character 28 (resp. 35) is the 4-th smallest, the second character 32 (resp. 40) is the 5-th smallest, the third character 12 (resp. 23) is the smallest, and so on. See Figure 1. The OPPM has been studied in the fields of finding some patterns affected by relative orders, not by their absolute values. For example, it can be applied to time series analysis like share prices on stock markets and to musical melody matching of two musical scores [1].

Recently, several results were presented on the OPPM problem. For the OPPM problem, the order-isomorphism must be defined. Kim et al. [1] defined the order-isomorphism as the equivalence of permutations converted from strings with an

⋆ Corresponding author.

P. Widmayer, Y. Xu, and B. Zhu (Eds.): COCOA 2013, LNCS 8287, pp. 295–305, 2013.
© Springer International Publishing Switzerland 2013

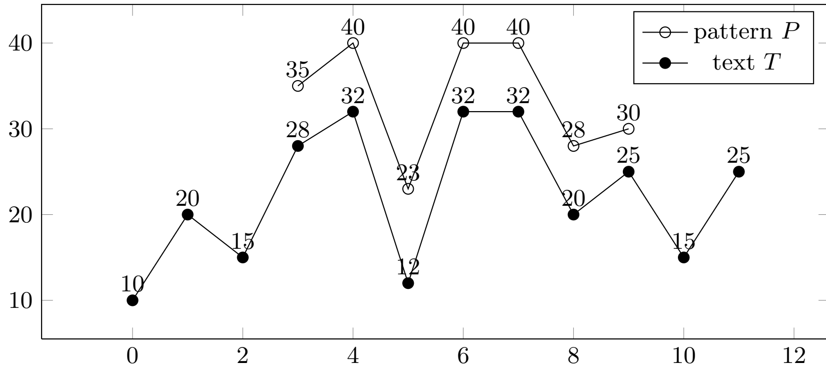

Fig. 1. An OPPM example for pattern $P = (35, 40, 23, 40, 40, 28, 30)$ and text $T = (10, 20, 15, 28, 32, 12, 32, 32, 20, 25, 15, 25)$

assumption that all the characters in a string are distinct. Given T ($|T| = n$) and P ($|P| = m$), they proposed an algorithm for the OPPM problem running in $O(n + m \log m)$ time based on the Knuth-Morris-Pratt (KMP) algorithm [4]. Meanwhile, Kubica et al. [2] defined the order-isomorphism as the equivalence of all relative orders between two strings, and presented a method of deciding the order-isomorphism of two strings even when there are same characters. They independently proposed an algorithm for the OPPM problem based on the KMP algorithm running in $O(n + m \log m)$ time for a general alphabet and $O(n + m)$ time for an integer alphabet. More recently, Crochemore et al. [3] introduced order-preserving suffix trees, and they suggested an algorithm finding all occurrences of P in T running in $O((m \log n)/ \log \log n + z)$ time where z is the number of occurrences.

In this paper, we propose a fast algorithm for the OPPM problem based on the Horspool algorithm [6–8]. Experimental results show that our algorithm is about 2 to 5 times faster than the KMP-based algorithm in reasonable cases. Our contributions are as follows.

- We present a new method of deciding the order-isomorphism between two strings even when there are same characters. We show that Kubica et al.'s method [2] may decide incorrectly when there are same characters.
- We show that the bad character rule can be applied to the OPPM problem by defining groups of characters as one character. Kim et al. [1] mentioned the hardness of applying the Boyer-Moore algorithm [5] to the OPPM problem. The good suffix rule could be well-defined but the bad character rule could not be directly applied to the OPPM problem.
- We present a space-efficient algorithm computing the shift table for text search based on a factorial number system. Let q be the size of the group of characters and $|\Sigma|$ be the size of the alphabet. Then, our algorithm uses $O(q!)$ space for the shift table while the algorithms of [6, 7] for the generic pattern matching problem use $O(|\Sigma|^q)$ space for the shift table.

Table 1. $LMax_P$, $LMin_P$, $\mu(P)$ for $P = (35, 40, 23, 40, 40, 28, 30)$

i	0	1	2	3	4	5	6
$P[i]$	35	40	23	40	40	28	30
$LMax_P[i]$	-1	0	-1	1	3	2	5
$LMin_P[i]$	-1	-1	0	1	3	0	0
$\mu(P)[i]$	0	1	0	3	4	1	2

This paper is organized as follows. In Section 2, we describe the previous works related to the OPPM problem. In Section 3, we present a new method of deciding the order-isomorphism between two strings. In Section 4, we present an algorithm for the OPPM problem. In Section 5, we show experimental results comparing our algorithm with the KMP-based algorithm.

2 Preliminaries

Let Σ denote an alphabet and $\sigma = |\Sigma|$. Let $|x|$ denote the length of a string x. A string x is described by a sequence of characters $(x[0], x[1], \ldots, x[|x| - 1])$. For a string x, let a substring $x[i..j]$ be $(x[i], x[i + 1], \ldots, x[j])$.

Now, we formally define the order-isomorphism and the order-preserving pattern matching problem. Two strings x and y of the same length over Σ are called *order-isomorphic*, written $x \approx y$, if

$$x[i] \leq x[j] \Leftrightarrow y[i] \leq y[j] \text{ for } 0 \leq i, j < |x|.$$

If two strings x and y are not order-isomorphic, we write $x \not\approx y$. Given a text $T[0..n - 1]$ and a pattern $P[0..m - 1]$, we say that T *matches* P at position i if $T[i - m + 1..i] \approx P$. In the previous example shown in Figure 1, T matches P at position 9 because $T[3..9] \approx P$. The *order-preserving pattern matching problem* is to find all positions of T matched with P.

Let us define a *prefix table* $\mu(x)$ of string x:

$$\mu(x)[i] = |\{j : x[j] \leq x[i] \text{ for } 0 \leq j < i\}|.$$

For the previous example, the prefix table of P is $\mu(P)[i] = (0, 1, 0, 3, 4, 1, 2)$. See Table 1.

Lemma 1. *For two strings x and y, if $x \approx y$, then $\mu(x) = \mu(y)$.*

Proof. By the assumption that $x \approx y$, $x[i] \leq x[j] \Leftrightarrow y[i] \leq y[j]$ for $0 \leq i < j < |x|$. Hence, $\mu(x) = \mu(y)$.

Lemma 2. *Assume that* $x[0..t] \approx y[0..t]$*. For all* $0 \le i, j \le t$*, if* $x[i] < x[j]$*, then* $y[i] < y[j]$*, and if* $x[i] = x[j]$*, then* $y[i] = y[j]$*.*

Proof. We first prove by contradiction the first proposition (when $x[i] < x[j]$). Suppose that $y[i] \ge y[j]$. Then, by the definition of order-isomorphism, $x[i] \ge x[j]$, which contradicts the assumption that $x[i] < x[j]$.

Next, consider the case when $x[i] = x[j]$. Then, since $x[i] \le x[j]$, $y[i] \le y[j]$ by the definition of order-isomorphism. Moreover, since $x[j] \le x[i]$, $y[j] \le y[i]$. Since $y[i] \le y[j]$ and $y[j] \le y[i]$, $y[i] = y[j]$.

Kubica et al. [2] used location tables called $LMax$ and $LMin$ for the order information of prefixes of P:
Given a string x, for $i = 0, \dots, |x| - 1$,

$$LMax_x[i] = j \text{ if } x[j] = \max\{x[k] : k \in [0, i-1], x[k] \le x[i]\};$$

if there is no such j then $LMax_x[i] = -1$. Similarly

$$LMin_x[i] = j \text{ if } x[j] = \min\{x[k] : k \in [0, i-1], x[k] \ge x[i]\},$$

and $LMin_x[i] = -1$ if no such j exists. If more than one such j exist, we select the rightmost one among them. Intuitively, $LMax_x[i]$ indicates the position of the largest character which is not larger than $x[i]$ in $x[0..i-1]$, and $LMin_x[i]$ indicates the position of the smallest character which is not smaller than $x[i]$ in $x[0..i-1]$. For the previous example, the location tables of P are $LMax_P[i] = (-1, 0, -1, 1, 3, 2, 5)$ and $LMin_P[i] = (-1, -1, 0, 1, 3, 0, 0)$. See Table 1. Notice the location tables of x can be computed in $O(|x|)$ time for an integer alphabet and in $O(|x| \log |x|)$ time for a general alphabet [2].

3 New Decision of Order-Isomorphism

In this section, we show that Kubica et al.'s method [2] for deciding the order-isomorphism of two strings may be incorrect when there are same characters and present a new method which works correctly even when there are same characters.

Kubica et al. [2] claimed that the order-isomorphism of two strings x and y could be decided using the location tables as follows.

Lemma 3 (see [2]). *Assume that* $x[0..t] \approx y[0..t]$*,* $t < |x| - 1, |y| - 1$ *and* $a = LMax_x[t+1]$*,* $b = LMin_x[t+1]$*. Then,* $x[0..t+1] \approx y[0..t+1] \Leftrightarrow y[a] \le y[t+1] \le y[b]$*. In case* a *or* b *is equal to* -1*, we omit the respective inequality in the condition.*

For example, assume two strings $x = (1, 3, 2)$, $y = (2, 5, 4)$, and the location tables $LMax_x = (-1, 0, 0)$ and $LMin_x = (-1, -1, 1)$ are given. Then, $y \approx x$ since $y[LMax_x[i]] \le y[i] \le y[LMin_x[i]]$ for all $0 \le i < 3$.

However, this method may decide incorrectly when there are same characters. For example, consider two strings $x = (1, 3, 2)$ and $y = (1, 2, 2)$.

Then, $y[LMax_x[i]] \leq y[i] \leq y[LMin_x[i]]$ for all $0 \leq i < 3$. But, by the definition of order-isomorphism, $y \not\approx x$ because $x[1] \not\leq x[2]$ and $y[1] \leq y[2]$. The reasons why Lemma 3 may not hold when there are same characters in the given strings are as follows. In the proof of the necessary condition of Lemma 3, to show $x[0..t+1] \approx y[0..t+1]$ (when $y[a] \leq y[t+1] \leq y[b]$), they tried to prove that $x[i] \leq x[t+1] \Leftrightarrow y[i] \leq y[t+1]$ for $i \leq t$. For this, they proved that $x[i] \leq x[t+1] \Rightarrow y[i] \leq y[t+1]$ and $x[i] \geq x[t+1] \Rightarrow y[i] \geq y[t+1]$. But, it is not equivalent to $x[i] \leq x[t+1] \Leftrightarrow y[i] \leq y[t+1]$. Instead of the latter $x[i] \geq x[t+1] \Rightarrow y[i] \geq y[t+1]$, it should be proven that $x[i] > x[t+1] \Rightarrow y[i] > y[t+1]$. As seen in our example, however, $x[1] > x[2] \not\Rightarrow y[1] > y[2]$.

We show a new lemma for deciding whether two strings are order-isomorphic or not even when there are same characters.

Lemma 4. *Assume that $x[0..t] \approx y[0..t]$, $t < |x|-1, |y|-1$ and $a = LMax_x[t+1]$, $b = LMin_x[t+1]$. Let p be the condition $y[a] < y[t+1]$ and q be the condition $y[t+1] < y[b]$. Then, $x[0..t+1] \approx y[0..t+1] \Leftrightarrow (p \wedge q)$ or $(\neg p \wedge \neg q)$. In case a or b is equal to -1, we assume the respective condition p or q is true.*

Proof. Without loss of generality, we assume that $a \neq -1$ and $b \neq -1$. Since $x[a] \leq x[b]$ by definitions of $LMax$ and $LMin$, $y[a] \leq y[b]$ by definition of the order-isomorphism. Hence, $(\neg p \wedge \neg q)$, i.e., $y[a] \geq y[t+1] \geq y[b]$ is equal to $y[a] = y[t+1] = y[b]$.

($\Rightarrow$) By definitions of $LMax$ and $LMin$, $x[a] \leq x[t+1] \leq x[b]$. We have two cases according to whether $x[a] = x[b]$ or not.

- Case when $x[a] = x[b]$: In this case, $x[a] = x[t+1] = x[b]$. Since $x[0..t+1] \approx y[0..t+1]$, $y[a] = y[t+1] = y[b]$ by Lemma 2.

- Case when $x[a] < x[b]$: First we prove that $x[a] \neq x[t+1] \neq x[b]$. Without loss of generality, suppose $x[t+1] = x[a]$. Then, $x[a]$ is the smallest character which is not smaller than $x[t+1]$ in $x[0..t+1]$. That is, $x[a] = x[b]$, which contradicts the condition that $x[a] < x[b]$. Since $x[a] \neq x[t+1] \neq x[b]$, $x[a] < x[t+1] < x[b]$ and thus $y[a] < y[t+1] < y[b]$ by Lemma 2.

Therefore, $x[0..t+1] \approx y[0..t+1] \Rightarrow (y[a] < y[t+1] < y[b])$ or $(y[a] = y[t+1] = y[b])$.

($\Leftarrow$) Since we have already $x[0..t] \approx y[0..t]$ (assumption), to show $x[0..t+1] \approx y[0..t+1]$, we only need to prove that for all $i \leq t$,

$$x[i] \leq x[t+1] \Leftrightarrow y[i] \leq y[t+1] \text{ and } x[t+1] \leq x[i] \Leftrightarrow y[t+1] \leq y[i].$$

We only consider the former, i.e., $x[i] \leq x[t+1] \Leftrightarrow y[i] \leq y[t+1]$. (The latter can be proven in a similar way.) First, we show that $x[i] \leq x[t+1] \Rightarrow y[i] \leq y[t+1]$ when $(p \wedge q)$ or $(\neg p \wedge \neg q)$. By the definition of $LMax$, $x[i] \leq x[a]$. Since $x[0..t] \approx y[0..t]$, $y[i] \leq y[a]$. Finally, by the hypothesis $(p \wedge q)$ or $(\neg p \wedge \neg q)$, $y[a] \leq y[t+1]$. Hence, we get $y[i] \leq y[t+1]$.

Next, we show that $y[i] \leq y[t+1] \Rightarrow x[i] \leq x[t+1]$ when $(p \wedge q)$ or $(\neg p \wedge \neg q)$. We have two cases according to the hypothesis $(p \wedge q)$ or $(\neg p \wedge \neg q)$.

- Case when $y[a] = y[t+1] = y[b]$ ($\neg p \wedge \neg q$): Since $y[i] \leq y[t+1] = y[a]$ and $x[0..t] \approx y[0..t]$, $x[i] \leq x[a]$. Moreover, since $y[a] = y[b]$, $x[a] = x[b]$ by Lemma 2, and then $x[t+1] = x[a] = x[b]$. Hence, $x[i] \leq x[a] = x[t+1]$.
- Case when $y[a] < y[t+1] < y[b]$ ($p \wedge q$): We prove it by contradiction. Suppose $x[i] > x[t+1]$. Then, $x[i] \geq x[b]$ by the definition of $LMin$, and thus $y[i] \geq y[b]$ due to $x[0..t] \approx y[0..t]$. Moreover, since $y[b] > y[t+1]$, we have $y[i] > y[t+1]$. It contradicts the condition that $y[i] \leq y[t+1]$.

Therefore, ($p \wedge q$) or ($\neg p \wedge \neg q$) $\Rightarrow x[0..t+1] \approx y[0..t+1]$.

□

For example, let us consider again the two strings $x = (1, 3, 2)$, $y = (1, 2, 2)$ and the location tables $LMax_x = (-1, 0, 0)$, $LMin_x = (-1, -1, 1)$ shown as the counter-example. Obviously, $x[0..1] \approx y[0..1]$ by the definition of the order-isomorphism. Then, $y \not\approx x$ because $y[LMax_x[2]] < y[2] = y[LMin_x[2]]$.

4 Fast Order-Preserving Pattern Matching Algorithm

4.1 Basic Idea

Basically, our algorithm for the OPPM problem is based on the Horspool algorithm widely used for generic pattern matching problems. The Horspool algorithm for generic pattern matching problems uses the shift table for filtering mismatched positions to expect sublinear behavior. (This method is well known as the bad character rule.) That is, when a mismatch occurs, the generic Horspool algorithm shifts the pattern using the shift table by setting the character of T compared with $P[m-1]$ as the bad character.

However, as mentioned in [1], it is not easy to apply the bad character rule to the OPPM problem since the order-isomorphism is defined using the orders of characters, not just the character itself. Consider the previous example again, i.e., $T = (10, 20, 15, 28, 32, 12, 32, 32, 20, 25, 15, 25)$ and $P = (35, 40, 23, 40, 40, 28, 30)$. If we apply the generic Horspool algorithm to this case, we should compare $T[6]$ with $P[6]$ first. $T[6] \approx P[6]$ by the definition of order-isomorphism but as we can see, $T[0..6] \not\approx P$. If we set $T[6]$ as the bad character as the generic Horspool algorithm, the shift table for $T[6]$ is hard to be defined since $T[6]$, no matter what character it is, will match every character in P by the definition of the order-isomorphism.

There exist some variants of the Horspool algorithm using the notion of q-grams which consider q consecutive characters as one character [6, 7]. When a mismatch occurs, the q-gram based algorithms shift the pattern farther than the original Horspool algorithm by some modification of the shift table. Given a q-gram x and a pattern P of length m over Σ, the shift table D in [6, 7] is defined as follows:

$$\text{Let } k = \max\{i \mid P[i-q+1..i] = x \text{ for } q-1 \leq i < m-1\}. \text{ Then,}$$

$$D[f(x)] = \min(m-q+1, m-k-1). \tag{1}$$

In (1), k means the last position of P matching a q-gram x. To index the shift table D, they defined a fingerprint $f(x)$ which maps a q-gram x to an integer. Intuitively, using $f(x)$, a q-gram x is mapped to a character over an alphabet whose size is σ^q. For a q-gram x, the fingerprint $f(x)$ is defined as follows.

$$f(x) = \sum_{k=0}^{q-1} x[k] \cdot \sigma^k$$

We use q-grams to solve the hardness of defining bad characters in the OPPM. For this, we should modify the shift table and the fingerprint. Given a q-gram x and a pattern P of length m, we define the shift table D indexed by the fingerprint $f(x)$ as follows:

Let $k = \max\{i \mid \mu(P[i-q+1..i]) = \mu(x) \text{ for } q-1 \leq i < m-1\}$. Then,

$$D[f(x)] = \min(m-q+1, m-k-1). \tag{2}$$

In (2), the meaning of k is the same as in (1), but we find the position of P matching a q-gram using the prefix table and a new fingerprint for space-efficiency of the shift table. Note that even if we use the prefix table instead of the location tables, we do not miss any position of P that matches the q-gram x by Lemma 1. We use a factorial number system [9] for our new fingerprint. Note that we can use the factorial number system since there are $i+1$ possible values for the i-th element of the prefix table. Refer to [9–11] for more details. For a q-gram x, we define a fingerprint $f(x)$ as follows.

$$f(x) = \sum_{k=0}^{q-1} \mu(x)[k] \cdot k! \tag{3}$$

Since the fingerprint $f(x)$ in (3) has the factorial number system, the prefix tables are uniquely mapped to integers from 0 to $q!-1$ [9–11]. Thus, our shift table D needs $O(q!)$ space.

4.2 Search Algorithm

Our algorithm consists of two steps. In the first step, we compute the location tables $LMax_P$, $LMin_P$ and the shift table D of pattern P. As mentioned above, the location tables can be computed in $O(m \log m)$ time for a general alphabet and can be computed in $O(m)$ time for an integer alphabet [2]. To compute D, all the fingerprints of q-grams of P must be computed. For all the q-grams of P, prefix tables can be computed in $O(m\,q \log q)$ time using dynamic order-statistics trees [1] for a general alphabet and can be computed in $O(m\,q)$ time using word-encoded sets [11] for an integer alphabet where $\sigma = 2^{\lfloor w/q \rfloor - 1}$ and w is the word size. Then, after computing all the prefix tables, all the fingerprints can be computed in $O(m\,q)$ time by Horner's rule [4]. Finally, D can be computed in $O(q! + m\,q \log q)$ time [6,7]. Note that we need $O(q!)$ time for initialization of D. The first step takes $O(q! + m\,q \log q + m \log m)$ for a general alphabet.

Algorithm 1 shows a pseudo-code of the second step, where we search for P in T using the shift table D. Suppose we check if P matches $T[i-m+1..i]$. We first compare the last q-grams of P and $T[i-m+1..i]$ using their fingerprints, i.e., $f(P[m-q..m-1])$ and $f(T[i-q+1..i])$. If they are the same, we check the order-isomorphism of P and $T[i-m+1..i]$ character by character using $LMax_P$ and $LMin_P$ (Lemma 4). Otherwise, we do not compare P and $T[i-m+1..i]$ because $T[i-m+1..i]$ cannot be order-isomorphic to P by Lemma 1. Then, we shift P forward by $D[f(T[i-q+1..i])]$. We repeat this until P reaches the rightmost of T. Figure 2 shows a part of process of Algorithm 1 on the previous example shown in Figure 1. We first compare the fingerprints $f(T[4..6]) = 4$ and $f(P[4..6]) = 2$. Since they are distinct, we shift P by $D[f(T[4..6])] = D[4] = 3$. Next, since $f(T[7..9])$ and $f(P[4..6])$ are the same, we compare P and $T[3..9]$ using Lemma 4. Since $P \approx T[3..9]$, Algorithm 1 reports the position 9 as an occurrence. Since the second step takes $O(nm + nq\log q)$ time for a general alphabet, Algorithm 1 takes $O(nm + nq\log q + q!)$ time overall. For an integer alphabet of size $\sigma = 2^{\lfloor w/q \rfloor - 1}$ where w is the word size, Algorithm 1 takes $O(nm + nq + q!)$ time.

Algorithm 1. Text Search

```
1: Preprocess D, LMax_P, LMin_P
2: m ← |P|, n ← |T|
3: t ← f(P[m − q..m − 1])
4: i ← m − 1
5: while i < n do
6:     c ← f(T[i − q + 1..i])
7:     if c = t then                              ▷ Compare the last q-grams
8:         if T[i − m + 1..i] ≈ P then
9:             print "pattern occurs at position" i
10:    i ← i + D[c]                               ▷ Shift P by D[c]
```

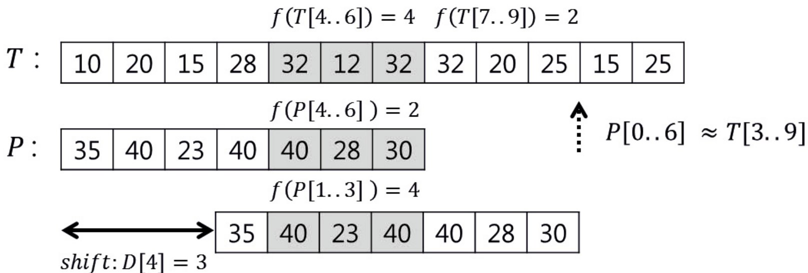

Fig. 2. Performing search on $T = (10, 20, 15, 28, 32, 12, 32, 32, 20, 25, 15, 25)$ with $P = (35, 40, 23, 40, 40, 28, 30)$ using Algorithm 1

Algorithm 2. Fingerprint Computation

$q \leftarrow |x|$, $c \leftarrow 0$
for $i \leftarrow q-1$ **downto** 1 **do**
  $t \leftarrow 0$
  for $j \leftarrow 0$ **to** $i-1$ **do**
    if $x[j] \leq x[i]$ **then** $t \leftarrow t+1$ ▷ Compute $\mu(x)[i]$
  $c \leftarrow (c+t) \cdot i$ ▷ Horner's rule
return c

5 Experimental Results

Table 2. Search times (in seconds) for 1,000 random patterns in a random text of length 5,000,000

σ	m	5			10			15		
	q	3	4	5	3	4	5	3	4	5
2^{30}	OKMP	41.76			41.78			41.84		
	OHq	**28.81**	**39.31**	82.17	**17.22**	**13.17**	**14.79**	**15.49**	**8.86**	**8.71**
10	OKMP	41.17			41.28			41.22		
	OHq	**28.75**	**39.50**	82.57	**17.39**	**13.26**	**14.82**	**15.79**	**8.99**	**8.75**
4	OKMP	41.43			41.28			41.29		
	OHq	**30.92**	**40.89**	83.18	**18.55**	**14.20**	**15.24**	**16.86**	**9.86**	**9.11**
2	OKMP	40.46			41.10			40.90		
	OHq	**37.99**	47.08	86.56	**24.55**	**19.41**	**18.60**	**21.72**	**14.21**	**11.67**

We conducted experiments to compare the practical performance of our algorithm (OHq) and the KMP-based algorithm (OKMP). The KMP-based algorithm was implemented based on the algorithms of [1,2]. We checked the order-isomorphism using Lemma 4 in both algorithms. We used a naive approach (Algorithm 2) to compute the fingerprints instead of using dynamic order-statistics trees or word-encoded sets because they are less practical when implemented. Algorithm 2 runs in $O(q^2)$ time.

The experimental environments and parameters are as follows. Both algorithms were implemented in C++ and compiled with Microsoft's C/C++ compiler (x86) version 17.00.50727.1, and O2 (maximizing speed) and Oi (generating intrinsic functions) options were used as optimization options. The experiments were performed on a Windows 7 PC (64bit) with 32 GB RAM and Intel Core i7 3820 processor. We tested for a random text T of length $n = 5,000,000$ from an integer alphabet and searched for 1,000 random patterns of length $m = 5, 10, 15$, respectively. We performed experiments with varying q from 3 to 5 and $\sigma = 2^{30}, 10, 4, 2$.

Table 2 shows search times. As the pattern length m becomes longer, OHq runs faster compared to OKMP. Especially, for example, when $\sigma = 2^{30}$, $m = 15$, and $q = 5$, OHq is about 5 times faster than OKMP. Whereas when $m = 5$, OHq does not work well compared to OKMP. The reason why OHq is relatively slower in this case is because it is based on the Horspool algorithm which works better as patterns are longer and σ is larger. When $m = 5$ and $q = 5$, OKMP beats OHq for all cases because $q = m$ and q-gram technique has no effect on speedup. From our experiment, it seems that setting $q = 4$ is adequate for short patterns ($m \leq 15$). Also, it is worthy of remark that the search times for each algorithm are almost the same regardless of the alphabet size. That is, the alphabet size hardly affects the search time in the order-preserving pattern matching.

Acknowledgments. This work was supported by the National Research Foundation of Korea (NRF) grant funded by the Korea government (MSIP) (No. 2012R1A2A2A01014892). This work was supported by the IT R&D program of MSIP/KEIT [10038768, The Development of Supercomputing System for the Genome Analysis]. This work was supported by the Industrial Strategic technology development program (10041971, Development of Power Efficient High-Performance Multimedia Contents Service Technology using Context-Adapting Distributed Transcoding) funded by the Ministry of Knowledge Economy (MKE, Korea). This research was supported by Basic Science Research Program through the National Research Foundation of Korea (NRF) funded by the Ministry of Science, ICT & Future Planning (2011-0007860). This research was supported by Next-Generation Information Computing Development Program through the National Research Foundation of Korea (NRF) funded by the Ministry of Science, ICT & Future Planning (2011-0029924).

References

1. Kim, J., Eades, P., Fleischer, R., Hong, S., Iliopoulos, C.S., Park, K., Puglisi, S.J., Tokuyama, T.: Order preserving matching. CoRR, abs/1302.4064 (2013); Submitted to Theor. Comput. Sci.
2. Kubica, M., Kulczynski, T., Radoszewski, J., Rytter, W., Walen, T.: A linear time algorithm for consecutive permutation pattern matching. Information Processing Letters 113(12), 430–433 (2013)
3. Crochemore, M., Iliopoulos, C.S., Kociumaka, T., Kubica, M., Langiu, A., Pissis, S.P., Radoszewski, J., Rytter, W., Walen, T.: Order-preserving suffx trees and their algorithmic applications. CoRR, abs/1303.6872 (2013)
4. Cormen, T.H., Leiserson, C.E., Rivest, R.L., Stein, C.: Introduction to Algorithms, 3rd edn. The MIT Press (2009)
5. Boyer, R.S., Moore, J.S.: A fast string searching algorithm. Comm. ACM 20(10), 762–772 (1977)
6. Baeza-Yates, R.: Improved string searching. Software: Practice and Experience 19(3), 257–271 (1989)

7. Tarhio, J., Peltola, H.: String matching in the DNA alphabet. Software: Practice and Experience 27(7), 851–861 (1997)
8. Horspool, R.N.: Practical fast searching in strings. Software: Practice and Experience 10(6), 501–506 (1980)
9. Knuth, D.E.: The Art of Computer Programming, 3rd edn. Seminumerical Algorithms, vol. 2. Addison-Wesley (1997)
10. Myrvold, W., Ruskey, F.: Ranking and unranking permutations in linear time. Information Processing Letters 79(6), 281–284 (2001)
11. Mares, M., Straka, M.: Linear-time ranking of permutations. Algorithms-ESA, 187–193 (2007)