Analysis of Modified Shell Sort for Fully Homomorphic Encryption*

Joon-Woo Lee[1], Young-Sik Kim[2], and Jong-Seon No[1]

[1] Seoul National University, Republic of Korea
[2] Chosun University, Republic of Korea

Abstract. The Shell sort algorithm is one of the most practically effective sorting algorithms. However, it is difficult to execute this algorithm with its intended running time complexity on data encrypted using fully homomorphic encryption (FHE), because the insertion sort in Shell sort has to be performed by considering the worst-case input data. In this paper, in order for the sorting algorithm to be used on FHE data, we modify the Shell sort with an additional parameter α and a gap sequence of powers of two. The modified Shell sort is found to have the trade-off between the running time complexity of $O(n^{3/2}\sqrt{\alpha + \log\log n})$ and the sorting failure probability of $2^{-\alpha}$. Its running time complexity is close to the intended running time complexity of $O(n^{3/2})$ and the sorting failure probability can be made very low with slightly increased running time. Further, the optimal window length of the modified Shell sort is also derived via convex optimization. The proposed analysis of the modified Shell sort is numerically confirmed by using a randomly generated arrays. Further, the performance of the modified Shell sort is numerically compared with the case of Ciura's optimal gap sequence and the case of the optimal window length obtained through the convex optimization.

Keywords: Fully Homomorphic Encryption · Insertion Sort · Shell Sort · Sorting Failure Probability.

1 Introduction

Fully homomorphic encryption (FHE) is an encryption scheme that provides encrypted data to an evaluation algorithm, which enables addition or multiplication of plaintext without decryption. FHE enables specific operations to be performed on encrypted information without leaking any clue to the plaintext. The notion of FHE was suggested by Rivest, Adleman, and Dertouzos in 1978 [10]. Although several cryptography researchers had attempted to construct the FHE scheme because of its effectiveness with respect to operations in cloud systems, no one had been able to successfully construct it until 2009, when

* This work was supported by Institute for Information & communications Technology Promotion (IITP) grant funded by the Korea government (MSIT) (R-20160229-002941, Research on Lightweight Post-Quantum Crypto-systems for IoT and Cloud Computing).

Gentry succeeded in developing an FHE scheme using an ideal lattice [8]. Several researchers suggested different types of FHE algorithms in series using the bootstrapping technique in Gentry's scheme and optimized the FHE schemes. Recently, the computation time of certain FHE schemes has been significantly reduced, which makes this scheme practically applicable. Further, the algorithms used on FHE data are expected to demonstrate the oblivious property, i.e., providing the most appropriate outputs without knowing any information about the input. In other words, the behavior of an oblivious algorithm does not depend on the input data. If it depends on the input, it implies the leakage of input information. The oblivious property of an algorithm is essential for FHE schemes to ensure privacy.

However, the bottleneck of operations on the FHE data is not in addition and multiplication, but involves non-arithmetic algorithms, such as sorting, searching, and neural networks, which are not frequently analyzed or optimized on FHE data. When processing large amounts of ciphertexts in cloud systems, it is frequently required to process the sorted data rather than unaligned data. Thus, one of the most essential non-arithmetic operations on the FHE data is the sorting algorithm, which is generally used as a subroutine algorithm of many algorithms. However, most sorting algorithms are not suitable for FHE data. For example, as the quick sort algorithm, which is one of the most popularly used sorting algorithms, is not an oblivious algorithm, it cannot be used on FHE data. Although numerous studies have been conducted to render the quick sort algorithm oblivious, its running time complexity becomes $O(n^2)$. Its actual running time is even longer than that of the bubble sort, which is considered to have the longest running time among all the known sorting algorithms. Therefore, modifying conventional sorting algorithms to make them suitable for FHE data is necessary. Several studies have been conducted for this purpose [1,3,6].

The Shell sort [11], which is one of the oldest sorting algorithms, is the generalized version of the insertion sort. The Shell sort algorithm is an in-place algorithm, which is fast and easy to implement, and thus, many systems use it as a sorting algorithm. It is known that Shell sort uses insertion sort as a subroutine algorithm, and insertion sort can be performed on the FHE data [2,3]. However, the Shell sort should be modified to be used in the FHE setting. If we do not allow any error in sorting, then insertion sort is expected to be quite conservative, i.e., the number of operations for sorting must be set for the worst case, because the insertion sort algorithm in the FHE setting is an oblivious algorithm. Thus, if we use insertion sort in the Shell sort, the running time complexity of Shell sort in the FHE setting must be $O(n^2)$, which makes the use of Shell sort ineffective. Therefore, it is important to devise a sorting algorithm that is better than the Shell sort on the FHE data in terms of running time complexity.

It is known [2] that we can reduce the running time of insertion sort on the FHE data by allowing a sorting failure probability (SFP) using what is known as the window technique. According to this technique, in each insertion sort, instead of inserting the ith element into the subarray of $(a_1, a_2, \cdots, a_{i-1})$, we insert the ith element into the subarray $(a_{i-k}, a_{i-k+1}, \cdots, a_{i-1})$ of length k, called the window length, immediately to the left of the ith element. We call this subarray a "window" with window length k.

In this paper, we devise a method to modify the Shell sort in the FHE setting using the window technique, with a running time complexity close to its original value of $O(n^{3/2})$, by deriving the running time complexity while taking the trade-off with the SFP for gap sequence 2^h. It is referred to as a "modified Shell sort". To this end, we use the exact distribution of window lengths of subarrays in each gap for successful sorting in the Shell sort. If the length of the subarray for the insertion sort in some gap is s, it is discovered that the average of the required window length for successful sorting is proportional to $\sqrt{s}$, and the right tail of its probability distribution is very thin. In the sorting process, the window length is provided as a constant multiple of $\sqrt{s}$, which ensures a negligible SFP. If the window length is close to $\beta\sqrt{s}$, the SFP decays as $e^{-\beta^2}$, which signifies a very fast-decaying function. Therefore, with a fixed negligible SFP, we can set a small window length so that the running time is asymptotically faster than that of the naive version of the Shell sort on FHE data. With this analysis, we can obtain the running time complexity $O(n^{3/2}\sqrt{\alpha + \log\log n})$ with SFP $2^{-\alpha}$. Thus, we can derive the trade-off between the running time complexity and the SFP using the window length and an additional parameter α.

In this paper, we address only the gap sequence of powers of two, i.e., $2^h, h = 1, 2, 3, \cdots$. Although this gap sequence is not optimal in terms of running time complexity, we first analyze the running time complexity of the modified Shell sort on FHE data, which is important for the FHE in cloud systems. The performance of the modified Shell sort is numerically compared with the cases of optimal window lengths obtained through convex optimization and Ciura's optimal gap sequence [5], which was evaluated numerically as an optimal gap sequence in non-FHE settings. Although we do not analyze this case, the method of deriving the optimal window length in the modified Shell sort functions well for Ciura's optimal gap sequence.

We also suggest the convex optimization method to derive a tighter window length. In other words, the window length obtained by the convex optimization method makes the running time of the modified Shell sort to be less than that of the case employing the analytical method in the modified Shell sort. This result is also numerically confirmed.

Thus, our contributions are summarized as follows:

– The exact distribution of required window length in each gap is obtained for successfully sorting each subarray in the Shell sort with the gap sequence of powers of two.
– We propose a modified Shell sort with the gap sequence of powers of two and an additional parameter α on FHE data, and derive its trade-off between the running time complexity $O(n^{3/2}\sqrt{\alpha + \log\log n})$ and the SFP $2^{-\alpha}$.
– The optimal window length of each gap in the modified Shell sort is derived via the convex optimization technique.

The remainder of this paper is organized as follows. Section 2 presents the preliminary of the paper, which includes the related sorting algorithms and the notion of FHE. In Section 3, we present the distribution of the required window length for each gap in the Shell sort on FHE data with the gap sequence of powers of two. Then, we propose a modified Shell sort for FHE and derive the trade-off between the running time complexity and the SFP. Section 4 discusses a method to deduce the optimal window length of each gap of the modified Shell sort using the convex optimization technique. Section 5 shows numerical results that support the proposed analysis. From these results, the performance in the case of the optimal gap sequence or the optimal window lengths can be observed. Section 6 concludes the study and discusses the scope for future research.

2 Preliminary

2.1 Fully Homomorphic Encryption

FHE is a public-key encryption scheme, which supports an arbitrary number of additions and multiplications of plaintext without decryption, so that anyone without the decryption key can operate the circuit with any ciphertext without leaking the information of its plaintext.

Gentry suggested the bootstrapping technique to transform homomorphic encryption to a certain degree [8], which allows only a finite number of operations on the FHE data. The bootstrapping operation has enabled several researchers to construct FHE schemes [8], which involves implementing the decryption circuit on encrypted data using the evaluation algorithm, that is, the addition and multiplication algorithms in FHE setting. All of the FHE schemes suggested thus far ensure security by adding some errors on a few elementary functions of plaintexts. As the addition and multiplication operations are repeated, the total number of errors increases, and if the total number of errors exceeds a certain limit, a decryption failure occurs. Thus, the errors need to be removed after a certain number of operations on the encrypted data, so that the ciphertexts can be further evaluated. The purpose of the bootstrapping operation is to reset the errors in the ciphertext when the errors are too large to be decrypted.

As bootstrapping utilizes a considerable amount of computation during the processing of FHE, the number of bootstrapping operations significantly affects the total operation time of FHE. In fact, the number of bootstrapping operations depends on the depth of the circuit. The lower the depth of a circuit, the fewer the number of bootstrapping operations. Thus, it is crucial to consider the number of the bootstrapping operations for each element, when bootstrapping is implemented in FHE schemes. If the total number of operations in an algorithm is fixed, it is better to evenly distribute the operations on the inputs. Furthermore, to stably address errors, deterministic algorithms are better than randomized algorithms. This is because we can predict the error size of each element in deterministic algorithms ensuring that these errors are handled easily and error control is optimized adequately.

2.2 Sorting Algorithms

Although there exist several sorting algorithms [9], we consider only the insertion sort and Shell sort in this paper. These are comparison-based sorting algorithms, which do not rely on the divide-and-conquer method.

The insertion sort is an iterative sorting algorithm that sorts from the left-most element. In each iteration, we define an element to be sorted into its left-side subarray as the pivot element. It is assumed that the elements to the left of the pivot element are already sorted. We then compare the already sorted elements with the pivot element, deduce its proper position, and insert it into this position. Its worst-case and average-case running time complexities are both $O(n^2)$. It is known that insertion sort is slightly faster than the bubble sort in practical cases.

The operations in the conventional insertion sort require the knowledge of its input, and this is not allowed in case of FHE data. Therefore, we cannot determine the correct position of a pivot element in the already sorted subarray in the FHE setting, and thus, the operation and behavior of the insertion sort needs to be modified. It is known [3] that we can perform an insertion sort on FHE data by sequentially swapping the pivot element with the elements in the already sorted subarray to its left, from left to right. In fact, the FHE version of the insertion sort has already been proposed, and its performance has been assessed numerically in the previous works [3, 6]. This operation, however, is inefficient, as the number of operations is always the same as that in the worst case, that is, its average-case running time complexity is estimated to be $O(n^2)$. This depreciates the value of the insertion sort in comparison with that of the bubble sort.

The Shell sort is a generalized version of the insertion sort. It requires a gap sequence, which is the decreasing sequence of a positive integer ending with 1. For each gap h and each integer j, $0 \leq j \leq h-1$, the $(hi+j)$-th elements $i = 0, 1, 2, \cdots$ are sorted using insertion sort. As the gap sequence ends with 1, we can finally obtain the correctly sorted array.

Even though the running time complexity of the Shell sort varies depending on the gap sequences, it is asymptotically better than that of insertion sort. To the best of our knowledge, a trial of the Shell sort on FHE data has not been performed thus far.

2.3 Comparison Operation in FHE

In the sorting algorithms in the FHE setting, the swap operation is performed by comparing two encrypted elements. Although it is not possible to determine the larger element in the FHE setting, it has been established that computing the maximum and minimum elements out of the two elements is possible in the FHE setting, even though these elements are encrypted.

Although bit-wise encrypted numbers can be compared using certain Boolean circuits, the circuit depth is so high that the number of required bootstrapping operations is too large. Therefore, such a comparison is quite inefficient. Recently, Cheon et al. proposed a numerical method for comparing homomorphically encrypted numbers [4]. They first suggested an efficient comparison for word-wise encrypted numbers. According to their method, the maximum function of two numbers a and b can be computed as

$$\max(a, b) = \frac{a+b}{2} + \frac{|a-b|}{2} = \frac{a+b}{2} + \frac{\sqrt{(a-b)^2}}{2}.$$

Then, the square root is approximated using Wilkes' algorithm, which is a two-variable iterative method. As Wilkes' algorithm includes only arithmetic operations, we can compute the square root of some values in the FHE setting.

Thus, the swap operation is achievable in case of both insertion and bubble sorts in the FHE setting.

3 Analysis of Modified Shell Sort over FHE

In this section, we propose a modified Shell sort using the window technique suggested in [2], and the probability distribution of the required window length for the successful sorting is also obtained. Finally, the running time complexity of the modified Shell sort in each gap for the successful sorting of each subarray is determined for FHE, considering the trade-off with the SFP.

3.1 A Modified Shell Sort over FHE

As insertion sort can be performed on FHE data, the Shell sort, which uses the insertion sort as a subroutine algorithm, can also be performed on FHE data. However, if the Shell sort is to be employed without any sorting failure in the FHE setting, it is expected to be considerably conservative. In other words, as we need to consider the worst case for each gap, its running time complexity becomes $O(n^2)$, which does not provide any advantage in comparison with a simple insertion sort. Thus, designing the Shell sort with a negligible SFP and a running time complexity close to the original average-case value of $O(n^{3/2})$ [7] is necessary.

To this end, we employ the window technique [2, 3] in the Shell sort. During the insertion sort in each gap, instead of searching the position of each element in the whole partially sorted array, we search for its position in the partially sorted subarray of a certain window length, located to the left of the pivot element, as shown in Fig. 1. Fig. 1 shows an example of the modified Shell sort using the window technique, where the gap is 4 and the window length is 2. Subarrays consisting of elements that are separated by the gap are sorted using the insertion sort. To sort each subarray, it is compared only with the elements that are located to its left, within a distance equal to the window length from the pivot element to be inserted, which is called the modified Shell sort.

The proposed modified Shell sort is described in Algorithm 1. As the minimum and maximum functions can be computed without knowing their plaintext in the FHE setting [4], neither of the operations in Algorithm 1 require any knowledge of the contents of elements in the array $A[i]$. Thus, Algorithm 1 can be executed in the FHE setting. In designing this algorithm, deciding the window length in each gap for successfully sorting each subarray in the Shell sort for the given SFP $2^{-\alpha}$ is not a trivial problem. Along with the design of the window length for each gap, we propose a modified Shell sort with an additional parameter α. Further, the running time complexity of the modified Shell sort is determined to be $O(n^{3/2}\sqrt{\alpha + \log\log n})$ with an SFP of $2^{-\alpha}$. The parameter α is determined only from the SFP, regardless of the input size n. In fact, α is considerably smaller than n and should be larger than or equal to $\sqrt{6}\log e - 1 \simeq 2.534$, the derivation of which is provided in a subsequent section of this paper. It is noted that the proposed modified Shell sort considers the trade-off between the running time complexity and the SFP.

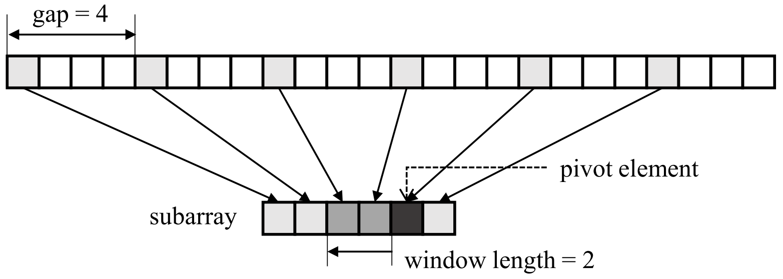


Fig. 1: Modified Shell sort using the window technique.

Algorithm 1: ModifiedShellSort($A[1:n], \alpha$)

Input : An array $A[1:n]$ with n elements and $\alpha \geq \sqrt{6}\log e - 1 \simeq 2.534$
Output: Sorted array $B[1:n]$ with SFP $2^{-\alpha}$

```
1  $c \leftarrow \alpha + 1 + \log\log n$
2  $p \leftarrow \lfloor \log n \rfloor$
3  for $\ell \leftarrow p$ to 0 do
4      $g \leftarrow 2^{\ell}$
5      $k \leftarrow \min\left\{\left\lceil \sqrt{\lceil \frac{n}{2g} \rceil \cdot (c+\ell) \cdot \frac{1}{\log e}} \right\rceil, \left\lceil \frac{n}{2g} \right\rceil\right\}$
6      for $i \leftarrow 2$ to $n$ do
7          $u \leftarrow \min\left\{k, \lceil \frac{i}{g} \rceil - 1\right\}$
8          for $j \leftarrow u$ to 1 do        // swapping of $A[i]$ and $A[i-gj]$
9              $d_1 \leftarrow A[i], d_2 \leftarrow A[i-gj]$
10             $A[i-gj] \leftarrow \min\{d_1, d_2\}$
11             $A[i] \leftarrow \max\{d_1, d_2\}$
12         end
13     end
14 end
```

3.2 Probability Distribution of Required Window Length

In this subsection, the probability distribution of required window length in each gap required for successfully sorting each subarray in the Shell sort is derived, as shown in Fig. 3. This probability distribution is essential in determining the window length of each gap in the modified Shell sort, because the properties of the tail of the probability distribution must be used to obtain the required window length.

While the analysis of the conventional Shell sort is performed for an average number of operations, the analysis of the window length in the modified Shell sort involves the maximum number of insertion operations for each subarray.

We assume that the gap sequence is powers of two, i.e., $2^{\lfloor \log n \rfloor}, 2^{\lfloor \log n \rfloor - 1}, \cdots, 2^2, 2, 1$. With this gap sequence, each subarray which is sorted using insertion sort has the following structure. The elements in odd positions of the subarray for a gap 2^h are already sorted and the elements in even positions are also sorted for a gap 2^h using the previous insertion sort for a gap 2^{h+1}. We analyze the insertion sort under this special situation.

The array of n elements is denoted by its index vector $(a_1, a_2, \cdots, a_n)$, which is a permuted vector of $(1, 2, \cdots, n)$. If we handle the real data, we map each datum to its respective index in $\{1, 2, \cdots, n\}$. Moreover, we assume that n is an even integer. If n is odd, the same analysis can be applied, with an additional dummy element inserted in the rightmost position with the largest element. Several lemmas are needed for devising the main theorem of the probability distribution for the required window length.

Lemma 1. *Let $\mathbf{a} = (a_1, a_2, \cdots, a_{2m})$ be a subarray in each gap in the Shell sort with a gap sequence 2^h, which is permuted from $(1, 2, \cdots, 2m)$, satisfying $a_i < a_{i+2}$ for $i = 1, 2, \cdots, 2m-2$. Let $M(\mathbf{a}) = \max_{1 \le i \le 2m} |a_i - i|$. Then there exists an even integer j and an odd integer k such that*

$$M(\mathbf{a}) = |a_j - j| = |a_k - k|$$

and $(a_j - j)(a_k - k) \le 0$.

Proof. Let M_1, M_2, M_3, and M_4 be defined as

$$\begin{aligned} M_1 &= \max_{1 \le i \le m} (a_{2i-1} - (2i-1)) \\ M_2 &= -\min_{1 \le i \le m} (a_{2i-1} - (2i-1)) \\ M_3 &= \max_{1 \le i \le m} (a_{2i} - 2i) \\ M_4 &= -\min_{1 \le i \le m} (a_{2i} - 2i). \end{aligned}$$

It is clear that at least one of M_1 and M_2 as well M_3 and M_4 is a non-negative integer. If we establish that $M_1 = M_4$ and $M_2 = M_3$, the lemma can be proved by the following argument. If $M_1 \ge M_2$, we obtain $M(\mathbf{a}) = M_1 = M_4 \ge M_2 = M_3$, and thus, there exist an odd index j and an even index k, such that $M(\mathbf{a}) =$

$a_j - j = -(a_k - k)$. If $M_1 < M_2$, $M(\mathbf{a}) = M_2 = M_3 \geq M_1 = M_4$ holds, then there exist an odd index j and an even index k, such that $M(\mathbf{a}) = -(a_j - j) = a_k - k$. Thus, it is sufficient to prove that $M_1 = M_4$ and $M_2 = M_3$.

To show $M_1 = M_4$ and $M_2 = M_3$, we prove the following four inequalities; $M_1 \geq M_4$, $M_1 \leq M_4$, $M_2 \geq M_3$, and $M_2 \leq M_3$.

i) Firstly, we show that $M_1 \geq M_4$. Consider an index l, such that $a_{2l} - 2l = \min_{1 \leq i \leq m}(a_{2i} - 2i)$, which is $-M_4$. We establish this case for $a_{2\ell} = 2m$ or $a_{2\ell} < 2m$.
  - i)-1 If $a_{2l} = 2m$, l must be m, as $2m$ is the largest element. Thus, we obtain $\min_{1 \leq i \leq m}(a_{2i} - 2i) = 0$ and $a_{2i} \geq 2i$ for all i, $1 \leq i \leq m$, which implies that 1 cannot be in the even index and must be in the first index, and $a_1 - 1 = 0$. Therefore, $M_1 = \max_{1 \leq i \leq m}(a_{2i-1} - (2i-1)) \geq 0 = -\min_{1 \leq i \leq m}(a_{2i} - 2i) = M_4$.
  - i)-2 If $a_{2l} < 2m$, we show that $a_{2l} + 1$ must be in the odd index. Let $a_{2l} + 1$ be in the even index; this implies that $a_{2l} + 1 = a_{2l+2}$, because all the elements in the even indices are already sorted. Then, we obtain $a_{2l+2} - (2l+2) = (a_{2l} + 1) - (2l+2) = a_{2l} - 2l - 1 < a_{2l} - 2l$, which is a contradiction to the assumption that $a_{2l} - 2l$ is the minimum value, and thus, $a_{2l} + 1$ must be in the odd index. Among $\{1, 2, \cdots, a_{2l} - 1\}$, $l-1$ elements have to be placed in the even indices in the left-side of a_{2l}. The remaining $a_{2l} - l$ elements must be placed in the odd indices in the increasing order from the first index 1. Thus, the index of $a_{2l}+1$ must be $2(a_{2l}-l)+1$. As $a_{2(a_{2l}-l)+1} - (2(a_{2l}-l)+1) = (a_{2l}+1) - (2(a_{2l}-1)+1) = 2l - a_{2l}$, we obtain $M_1 = \max_{1 \leq i \leq m}(a_{2i-1} - (2i-1)) \geq 2l - a_{2l} = -\min_{1 \leq i \leq m}(a_{2i} - 2i) = M_4$.

ii) We then show that $M_2 \geq M_3$. Consider an index l, such that $a_{2l} - 2l = \max_{1 \leq i \leq m}(a_{2i} - 2i)$, which is M_3. We establish this case for $a_{2\ell} = 1$ or $a_{2\ell} > 1$.
  - ii)-1 If $a_{2l} = 1$, l must be 1, as 1 is the smallest element, and therefore, $\max_{1 \leq i \leq m}(a_{2i} - 2i) = -1$. As $a_{2i} - 2i \leq -1$ for all i, $1 \leq i \leq m$, $2m$ cannot belong to the even index. Thus, $2m$ must be in the $(2m-1)$-th index, and $a_{2m-1} - (2m-1) = 1$. Therefore, $M_2 = -\min_{1 \leq i \leq m}(a_{2i-1} - (2i-1)) \geq -1 = \max_{1 \leq i \leq m}(a_{2i} - 2i) = M_3$.
  - ii)-2 If $a_{2l} > 1$, we show that $a_{2l} - 1$ must be in the odd index. Let $a_{2l} - 1$ be in the even index. We have $a_{2l} - 1 = a_{2l-2}$, because all the elements in the even indices are already sorted. Then, we obtain $a_{2l-2} - (2l-2) = (a_{2l} - 1) - (2l-2) = a_{2l} - 2l + 1 > a_{2l} - 2l$, which is a contradiction to the assumption that $a_{2l} - 2l$ is the maximum value, and thus, $a_{2l} - 1$ must be in the odd index. Among $\{1, 2, \cdots, a_{2l} - 2\}$, $l-1$ elements need to be placed in the even indices in the left-side of a_{2l}. The remaining $a_{2l} - l - 1$ elements have to be placed in the odd indices from the first index 1. Thus, the index of $a_{2l} - 1$ must be $2(a_{2l} - l) - 1$. As $a_{2(a_{2l}-l)-1} - (2(a_{2l}-l)-1) = (a_{2l}-1) - (2(a_{2l}-l)-1) = 2l - a_{2l}$, we obtain $M_2 = -\min_{1 \leq i \leq m}(a_{2i-1} - (2i-1)) \geq -(2l - a_{2l}) = \max_{1 \leq i \leq m}(a_{2i} - 2i) = M_3$.

Similarly, we can establish that $M_1 \leq M_4$ and $M_2 \leq M_3$ by swapping the even indices with the odd indices. Therefore, we can prove that $M_3 = \max_{1\leq i\leq m}(a_{2i} - 2i) \geq -\min_{1\leq i\leq m}(a_{2i-1} - (2i-1)) = M_2$, and $M_4 = -\min_{1\leq i\leq m}(a_{2i} - 2i) \geq \max_{1\leq i\leq m}(a_{2i-1} - (2i-1)) = M_1$. Therefore, we establish that $M_1 = M_4$ and $M_2 = M_3$. □

Lemma 2. *Let* $\mathbf{a} = (a_1, a_2, \cdots, a_{2m})$ *be a subarray in the Shell sort with a gap sequence* 2^h*, which is permuted from* $(1, 2, \cdots, 2m)$ *satisfying* $a_i < a_{i+2}$ *for* $i = 1, 2, \cdots, 2m-2$*. Let* $W(\mathbf{a})$ *be the required minimum window length to sort the subarray successfully. Then, we have*

$$W(\mathbf{a}) = \max_{1\leq i\leq 2m}(i - a_i).$$

Proof. When we insert a_i into the partially sorted subarray, the following scenarios can be given; if $a_i < i$, we require a window length of $i - a_i$, and if $a_i \geq i$, a_i stays in place regardless of the window length.

Consider the first case, where $a_i < i$. First, we assume that i is even. Consider the elements to the left of a_i. From the condition $a_i < a_{i+2}$, it is clear that all the elements in even indices to the left of a_i are less than a_i. As there are $i/2 - 1$ even indices to the left of a_i, the remaining $a_i - i/2$ elements in $\{1, 2, \cdots, a_i - 1\}$ have to be placed in odd indices in increasing order from the leftmost odd index. As the number of odd indices to the left of a_i is $i/2$ and $i/2 > a_i - i/2$, all the elements less than a_i are located to the left of a_i.

We then assume that i is odd. The proof is almost the same as that for the scenario in which i is even. As there are $(i-1)/2$ odd indices to the left of a_i, the remaining $a_i - (i+1)/2$ elements in $\{1, 2, \cdots, a_i - 1\}$ must be placed in even indices from the first even index, in increasing order. As the number of even indices to the left of a_i is $(i-1)/2$ and $(i-1)/2 > a_i - (i+1)/2$, all the elements less than a_i are located to the left of a_i.

Thus, we prove that all the elements less than a_i are located to the left of a_i. The partially sorted subarray, therefore, must include the elements $\{1, 2, \cdots, a_i - 1\}$ in the indices $\{1, 2, \cdots, a_i - 1\}$ in the appropriate order. This implies that a_i moves to the index a_i, and thus, we require a minimum window length of $i - a_i$.

Consider the second case, in which $a_i \geq i$. It is evident that $i/2 \leq a_i - i/2$, when i is even, and $(i-1)/2 \leq a_i - (i+1)/2$, when i is odd. This implies that all the elements to the left of a_i are less than a_i. Thus, the partially sorted subarray in the indices $\{1, 2, \cdots, i-1\}$ comprises elements smaller than a_i. Therefore, a_i does not move to the left but stays in its position, regardless of the window length. □

From Lemma 1, it is noted that $M(\mathbf{a})$ is equal to $W(\mathbf{a})$.

Lemma 3. *Let* $p_k(n, m)$ *be the number of distinct arrays* $(a_1, a_2, \cdots, a_m)$ *of length* m*, whose elements from* $\{1, 2, \cdots, n\}$ *are sorted in increasing order,* $a_i < a_{i+1}$*, and* $\max_{1\leq i\leq m} |a_i - 2i| \leq k$ *is satisfied for a positive integer* k *and* $n \geq m$*. Let* $(b_0, b_1, \cdots)$ *and* $(c_0, c_1, \cdots)$ *be the two arrays defined as*

$$b_0 = c_0 = 0$$

$$b_{i+1} = \begin{cases} b_i + (k+1) & \textit{if } i \textit{ is even} \\ b_i + (k+2) & \textit{if } i \textit{ is odd} \end{cases}$$

$$c_{i+1} = \begin{cases} c_i + (k+2) & \textit{if } i \textit{ is even} \\ c_i + (k+1) & \textit{if } i \textit{ is odd.} \end{cases}$$

For $2m-k \le n \le 2m+k$, we obtain

$$p_k(n,m) = \binom{n}{m} - \sum_{1 \le b_i \le m} (-1)^{i+1} \binom{n}{m-b_i} - \sum_{1 \le c_i \le m} (-1)^{i+1} \binom{n}{m+c_i}. \quad (1)$$

Proof. It is clear that $p_k(1,1) = 1$ for all $k \ge 1$, and $p_k(n,1) = n$ for $n \le k+2$. As the element a_m in the last index must be $2m-k \le a_m \le 2m+k$ from $\max_{1 \le i \le m} |a_i - 2i| \le k$, the following can be determined from the condition $2m-k \le a_m \le 2m+k$:

i) For $n < 2m-k$,
$$p_k(n,m) = 0,$$
because the minimum possible value of a_m must be $2m-k$.

ii) For $n > 2m+k+1$,
$$p_k(n,m) = p_k(2m+k,m),$$
because the maximum possible value of a_m must be $2m+k$.

iii) For $2m-k \le n \le 2m+k+1$,
We derive the recurrence relation of $p_k(n,m)$ using the following three cases:

iii)-1 For $n = 2m+k+1$,
It is easy to derive that
$$p_k(2m+k+1,m) = p_k(2m+k,m). \quad (2)$$
Note that this case can be included in ii). Although this separation of the case appears unnatural, it enables us to analyze $p_k(n,m)$ well.

iii)-2 For $2m-k+1 \le n \le 2m+k$,
If the element in the last index m is n, the elements in the remaining indices should be selected from $\{1,2,\cdots,n-1\}$, and thus, there are $p_k(n-1,m-1)$ possible arrays. If the element in the last index m is not n, the element n cannot be located in one of the indices $\{1,2,\cdots,m-1\}$, because the elements are sorted in increasing order. Thus, $\{1,2,\cdots,n-1\}$ should be located in the indices $\{1,2,\cdots,m\}$, and there are $p_k(n-1,m)$ possible arrays. Therefore, we obtain
$$p_k(n,m) = p_k(n-1,m) + p_k(n-1,m-1). \quad (3)$$

iii)-3 For $n = 2m-k$,
We obtain
$$p_k(2m-k,m) = p_k(2m-k-1,m-1), \quad (4)$$
because the element $2m-k$ must be located in the index m.

We prove the lemma using these three cases and overlapped Pascal's triangles, as shown in Fig. 2. If we define $p_k(0,m)=0$ for $1 \le m \le k$ and $p_k(0,0)=1$, then all of $p_k(n,m)$ are well-defined. This relation is similar to the Pascal's triangle $\binom{n}{m}=\binom{n-1}{m}+\binom{n-1}{m-1}$ shown in Fig. 2(a), except that the width of the triangle for $p_k(n,m)$ is limited, as shown in Fig. 2(b) as well as (2) and (4). This recursive relation can then be transformed into overlapped Pascal's triangles. Fig. 2(c) shows a part of Fig. 2(b) near the boundary of the lower dotted line. Here, we only consider the lower dotted line. We then establish that this recursive relation near the boundary in Fig. 2(c) is equivalent to the situation of Fig. 2(d), which is two overlapped Pascal's triangles, in which $p_k(0,0)=1$ and $p_k(0,k+1)=-1$.

First, it can be obtained that the values on the dotted line in Fig. 2(d) are always 0, because of the symmetry of Pascal's triangles. As adding a 0 does not change the value, the cases of Fig. 2(c) and Fig. 2(d) are equivalent corresponding to the area to the left of the dotted line.

However, the values on both the dotted lines in Fig. 2(b) must be 0. To satisfy the other boundary condition $p_k(2m+k+2,m)=0$ on the upper dotted line in Fig. 2(b), we consider another Pascal's triangle translated by $-(k+2)$ with $p_k(0,-(k+2))=-1$. If we add these three Pascal's triangles P_{-1}, P_0, and P_1 shown in Fig. 2(e), there are zero boundary values on the lines from Q_1 to Q_2 and from R_1 to R_2. However, the boundary value after Q_2 or R_2 is not equal to 0. To obtain the boundary values on the lines from Q_2 to Q_3 and from R_2 to R_3, we must add the Pascal's triangles P_2 and P_{-2}. Therefore, we repeat this process, as shown in Fig. 2(e). The sequence $\{b_i\}$ in Lemma 3 is the distance from the initial vertex of P_0 to that of P_i, while $\{c_i\}$ is the distance from the initial vertex of P_0 to that of P_{-i}. The initial value at the initial vertex of P_i is 1 if i is even, and -1 if i is odd. Q_i is defined as the intersection of the boundaries of the two Pascal's triangles starting from the initial vertices of P_{i-1} and P_{-i}, and R_i is defined as the intersection of the boundaries of the two Pascal's triangles starting from the initial vertices of P_i and $P_{-(i-1)}$.

We establish that if the Pascal's triangles P_i's, $i=\cdots,-1,0,1,\cdots$, are overlapped, all of the integer points on the half-lines of $\overrightarrow{Q_1Q_2}$ and $\overrightarrow{R_1R_2}$ must be 0s. The integer points on the upper half-line of $\overrightarrow{Q_1Q_2}$ exhibit the form $n=2m+k+2$ for all non-negative integers m, and those on the lower half-line of $\overrightarrow{R_1R_2}$ exhibit the form $n=2m-k-1$ for all $m \ge k+1$. First, in the case of the points on the half-line of $\overrightarrow{Q_1Q_2}$, we consider the integer points on $\overline{Q_jQ_{j+1}}$, which can be denoted as $n_1=2m_1+k+2$ and $b_{i-1} \le m_1 \le b_i$. Then, we can only consider Pascal's triangles $P_{-j},\cdots,P_{j-1}$. Considering the parallel translation of each Pascal's triangle, the overlapped values on the points are defined as

$$\sum_{i=1}^{j}(-1)^i\binom{2m_1+k+2}{m_1+c_i}+\sum_{i=1}^{j}(-1)^{i-1}\binom{2m_1+k+2}{m_1-b_{i-1}}. \tag{5}$$

As $(m_1+c_i)+(m_1-b_{i-1})=2m_1+k+2$, $\binom{2m_1+k+2}{m_1+c_i}=\binom{2m_1+k+2}{m_1-b_{i-1}}$ holds, and (5) is equal to 0.

In the case of the points on the half-line of $\overrightarrow{R_1R_2}$, we consider the integer points on $\overline{R_jR_{j+1}}$, which can be denoted as $n_2 = 2m_2 - k - 1$ and $b_i \leq b_{i+1}$. Then, we can only consider Pascal's triangles $P_{j-1}, \cdots, P_j$. The overlapped values on the points are defined as

$$\sum_{i=1}^{j} (-1)^{i-1} \binom{2m_2 - k - 1}{m_1 + c_{i-1}} + \sum_{i=1}^{j} (-1)^{i} \binom{2m_2 - k - 1}{m_1 - b_i}. \tag{6}$$

As $(m_2 + c_{i-1}) + (m_2 - b_i) = 2m_2 - k - 1$, $\binom{2m_2-k-1}{m_1+c_{i-1}} = \binom{2m_2-k-1}{m_1-b_i}$ holds, and (6) is also equal to 0.

Therefore, we establish that with respect to the region between the two dotted lines in Fig. 2(b), Fig. 2(b) is exactly equivalent to the hashed part of Fig. 2(e). We obtain $p_k(n, m)$ by adding the values of points of several Pascal's triangles as in (1), where the first term is from the central Pascal's triangle P_0; the second term is from the right-side Pascal's triangles P_i's for the positive integer i; and the third term is from the left-side Pascal's triangles P_{-i}'s for the positive integer i.

□

From the previous lemmas, we have the following theorem.

Theorem 4. *Let $C(2m, k)$ be the number of the permutations $\mathbf{a}$ of $\{1, 2, \cdots, 2m\}$, such that $a_i < a_{i+2}$ for all possible i, and $W(\mathbf{a}) \leq k$. Then, we have*

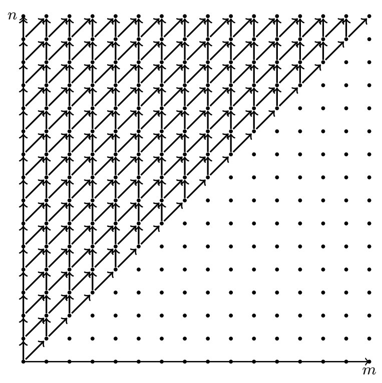
$$C(2m, k) = \binom{2m}{m} - \sum_{1 \leq b_i \leq m} (-1)^{i+1} \binom{2m}{m - b_i} - \sum_{1 \leq c_i \leq m} (-1)^{i+1} \binom{2m}{m - c_i}$$

where b_i and c_i are defined in Lemma 3.

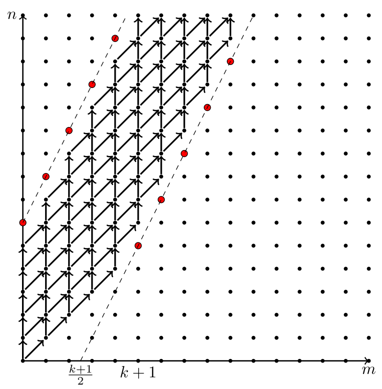
Proof. As $M(\mathbf{a})$ of the odd indices is equal to that of the even indices from Lemma 1, we consider only the even indices. Thus, we can consider this situation to be equivalent to the following simple situation; we consider distinct m elements from $\{1, 2, \cdots, 2m\}$ randomly, sort them in increasing order, and consider $a_i - 2i$ rather than $a_i - i$. Then, $C(2m, k)$ is identical to $p_k(2m, m)$ in Lemma 3. This is established as $\binom{2m}{m+b_i} = \binom{2m}{m-b_i}$.

□

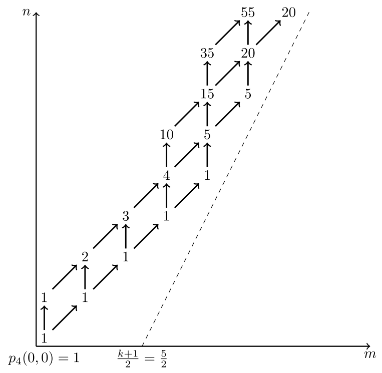
In fact, $C(2m, k)$ denotes the number of arrays for gap 2^h, which can be successfully sorted using the proposed modified Shell sort with a window length of k. Clearly, the exact number of arrays with $W(\mathbf{a}) = k$, such that $a_i < a_{i+2}$ for all i can be obtained by computing $C(2m, k) - C(2m, k-1)$. Fig. 3 shows the shape of the distribution of $C(2m, k) - C(2m, k-1)$. The peak of the curve is observed at 32, which is the approximated value of $\sqrt{1000}$. With this result, we derive the running time complexity of the modified Shell sort in the next subsection.



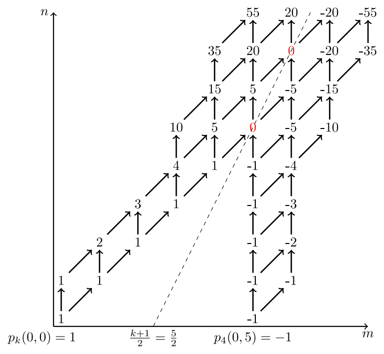
(a) Pascal's triangle for $\binom{n}{m}$



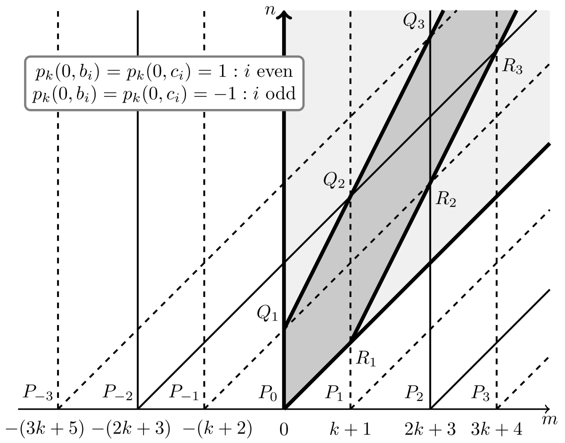
(b) $p_k(n,m)$ with $k=4$ for $2m-k \le n \le 2m+k$



(c) Boundary case of $p_k(n,m)$ in (b)



(d) Equivalent diagram of boundary of $p_k(n,m)$ in (b)



(e) $p_k(n,m)$ using overlapped Pascal's triangles

Fig. 2: $p_k(n,m)$ using Pascal's triangle.

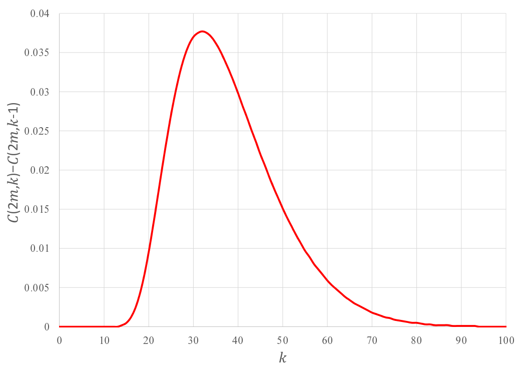


Fig. 3: Distribution of $C(2m,k) - C(2m,k-1)$ for $m = 1000$.

3.3 Derivation of Running Time Complexity for a Specific SFP

In this subsection, we derive the running time complexity $O(n^{3/2}\sqrt{\alpha + \log\log n})$ of the proposed modified Shell sort, considering the optimal trade-off with the SFP $2^{-\alpha}$, in which α is the parameter that controls the window length of each gap. In the running time complexity, $\log\log n$ increases gradually as n increases. Therefore, the running time complexity is approximately proportional to $n^{3/2}\sqrt{\alpha}$. However, the probability that the output is not successfully sorted decreases exponentially as α increases. It is noted that the SFP $2^{-\alpha}$ is not related to the input data size. One of the advantages of the modified Shell sort algorithm is irrespective of the size of the input data, and thus we can obtain a trade-off between the SFP and running time complexity by considering an appropriate α.

It is important to prove the following lemmas to determine the relation between the binomial coefficients and exponential function. It is a well-known fact from the central limit theorem in statistics that the closer n is to infinity, the closer a binomial distribution is to a normal distribution. Even though the binomial and normal distributions are similar, we should establish that some binomial coefficients are upper-bounded by the probability distribution function of the normal distribution. The following lemma is used in the proof of Lemma 6, and Lemma 6 is used to prove Theorem 7.

Lemma 5. *Let $f : [a, \infty) \to \mathbb{R}$ be a function of some real number a satisfying the following;*

i) $\lim_{x\to\infty} f(x) = M$ *for some real number M.*

ii) There exists a positive integer n, such that the n-th order derivative $f^{(n)}(x)$ exists on (a, ∞), and $(-1)^n f^{(n)}(x) > 0$ for all $x \in (a, \infty)$.

Then, $f(x) > M$ for all $x \in [a, \infty)$.

Proof. It is sufficient to show that $f^{(m)}(x) \to 0$ as $x \to \infty$ and $(-1)^m f^{(m)}(x)$ is a monotonically decreasing function for m, $1 \le m \le n-1$. If this is proved, then $f(x)$ is a monotonically decreasing function and is larger than the limit value M from the first condition in Lemma 5, as $f'(x)$ is negative for (a, ∞). Since it is true for $m = n$ that $(-1)^m f^{(m)}(x) > 0$, we will prove the following: if it is true for $2 \le k \le n$ that $(-1)^k f^{(k)}(x) > 0$, then we have $\lim_{x\to\infty} f^{(k-1)}(x) = 0$, and it is true that $(-1)^{k-1} f^{(k-1)}(x)$ is a monotonically decreasing function.

Let $g_k(x) = (-1)^k f^{(k)}(x)$. As $(-1)^{k-1} f^{(k)}(x) = g'_{k-1}(x) < 0$ on (a, ∞), $g_{k-1}(x)$ is a monotonically decreasing function. As a monotonically decreasing function always converges to a certain value, if it possesses some lower bound, we obtain $\lim_{x\to\infty} g_{k-1}(x) = T$ for some T, or $\lim_{x\to\infty} g_{k-1}(x) = -\infty$. We assume that $\lim_{x\to\infty} g_{k-1}(x) = T$ for some $T \neq 0$, or $\lim_{x\to\infty} g_{k-1}(x) = -\infty$. Then, we can deduce some $N \in (a, \infty), R > 0$, such that $|g_{k-1}(x)| > R$, i.e., $f^{(k-1)}(x) > R$ for all $x > N$, or $f^{(k-1)}(x) < -R$ for all $x > N$.

Consider the case of $f^{(k-1)}(x) > R$. If we integrate both terms from N to $x \in (N, \infty)$ iteratively as

$$f^{(k-2)}(x) - f^{(k-2)}(N) = \int_N^x f^{(k-1)}(t)dt > \int_N^x R dx = R(x-N)$$

$$\begin{aligned} f^{(k-3)}(x) - f^{(k-3)}(N) = \int_N^x f^{(k-2)}(t)dt &> \int_N^x \left(R(x-N) + f^{(k-2)}(N)\right) dx \\ &= \frac{R}{2}(x-N)^2 + f^{(k-2)}(N)(x-N), \end{aligned}$$

we obtain

$$f(x) > \frac{R}{(k-1)!}(x-N)^{k-1} + \sum_{i=0}^{k-2} \frac{f^{(i)}(N)}{i!}(x-N)^i,$$

whose right-hand side tends to infinity, as $x \to \infty$. In this case, $f(x)$ tends to infinity as well, which contradicts the first condition. If we consider the case of $f^{(m)}(x) < -R$, the inequality is changed to

$$f(x) < -\frac{R}{(k-1)!}(x-N)^{k-1} + \sum_{i=0}^{k-2} \frac{f^{(i)}(N)}{i!}(x-N)^i,$$

whose right-hand side tends to negative infinity, as $x \to \infty$. Then $f(x)$ tends to negative infinity as well, which also contradicts the first condition.

Thus, we obtain $\lim_{x\to\infty} g_{k-1}(x) = 0$. As $g_{k-1}(x)$ is a monotonically decreasing function, $g_{k-1}(x) > 0$ on (a, ∞), which completes the proof. □

Lemma 6. *For any real number $\alpha \ge \sqrt{6}$ and any positive integer $n \ge \lceil \alpha^2 \rceil$, the following inequality holds*

$$\binom{2n}{n - \lceil \alpha\sqrt{n} \rceil} < e^{-\alpha^2}\binom{2n}{n}.$$

Proof. It can be derived that

$$\frac{\binom{2n}{n}}{\binom{2n}{n-\lceil\alpha\sqrt{n}\rceil}} = \frac{(n+\lceil\alpha\sqrt{n}\rceil)(n+\lceil\alpha\sqrt{n}\rceil-1)\cdots(n+1)}{n(n-1)\cdots(n-\lceil\alpha\sqrt{n}\rceil+1)} = \prod_{k=0}^{\lceil\alpha\sqrt{n}\rceil-1}\left(1+\frac{\lceil\alpha\sqrt{n}\rceil}{n-k}\right).$$

We must prove that

$$\prod_{k=0}^{\lceil\alpha\sqrt{n}\rceil-1}\left(1+\frac{\lceil\alpha\sqrt{n}\rceil}{n-k}\right) > e^{\alpha^2}. \tag{7}$$

If we consider the logarithm on the left-hand side and change the form, we obtain

$$\sum_{k=0}^{\lceil\alpha\sqrt{n}\rceil-1}\ln\left(1+\frac{\lceil\alpha\sqrt{n}\rceil}{n-k}\right) \geq \sum_{k=0}^{\lceil\alpha\sqrt{n}\rceil-1}\ln\left(1+\frac{\alpha}{\sqrt{n}-\frac{k}{\sqrt{n}}}\right). \tag{8}$$

Let $f(x)=\log\left(1+\frac{\alpha}{x}\right)$. Then, the right-hand side of (6) can be defined as

$$\sqrt{n}\sum_{k=1}^{\lceil\alpha\sqrt{n}\rceil}\frac{1}{\sqrt{n}}f(\sqrt{n}+\frac{1}{\sqrt{n}}-\frac{k}{\sqrt{n}}),$$

which is a type of Riemann sum of $f(x)$. As $f(x)$ is a monotonically decreasing function, the Riemann sum demonstrates its lower bound as the integration of $f(x)$ from $\sqrt{n}+\frac{1}{\sqrt{n}}-\frac{\lceil\alpha\sqrt{n}\rceil}{\sqrt{n}}$ to $\sqrt{n}+\frac{1}{\sqrt{n}}$. As $\sqrt{n}+\frac{1}{\sqrt{n}}-\frac{\lceil\alpha\sqrt{n}\rceil}{\sqrt{n}} \leq \sqrt{n}+\frac{1}{\sqrt{n}}-\alpha$, we obtain

$$\begin{aligned}\sum_{k=0}^{\lceil\alpha\sqrt{n}\rceil-1}\ln\left(1+\frac{\alpha}{\sqrt{n}-\frac{k}{\sqrt{n}}}\right) &\geq \sqrt{n}\int_{\sqrt{n}+\frac{1}{\sqrt{n}}-\frac{\lceil\alpha\sqrt{n}\rceil}{\sqrt{n}}}^{\sqrt{n}+\frac{1}{\sqrt{n}}}\ln\left(1+\frac{\alpha}{x}\right)dx \\ &\geq \sqrt{n}\int_{\sqrt{n}+\frac{1}{\sqrt{n}}-\alpha}^{\sqrt{n}+\frac{1}{\sqrt{n}}}\ln\left(1+\frac{\alpha}{x}\right)dx. \end{aligned}\tag{9}$$

To integrate right-hand side of (9), let $g(x)=x\ln x$. We then obtain

$$\sqrt{n}\int_{\sqrt{n}+\frac{1}{\sqrt{n}}-\alpha}^{\sqrt{n}+\frac{1}{\sqrt{n}}}\ln\left(1+\frac{\alpha}{x}\right)dx = g(n+\alpha\sqrt{n}+1)+g(n-\alpha\sqrt{n}+1)-2g(n+1).$$

Let $h(x)=g(x^2+\alpha x+1)+g(x^2-\alpha x+1)-2g(x^2+1)$ in $[\alpha,\infty)$. If we prove $\lim_{x\to\infty}h(x)=\alpha^2$, and $h^{(3)}(x)<0$ in (α,∞), we obtain $h(x)>\alpha^2$ in $[\alpha,\infty)$ using Lemma 5. As $\sqrt{n}\geq\alpha$, we obtain $h(\sqrt{n})>\alpha^2$, which proves (7). We must establish $\lim_{x\to\infty}h(x)=\alpha^2$, and $h^{(3)}(x)<0$ in (α,∞). To prove $\lim_{x\to\infty}h(x)=\alpha^2$, we consider $e^{h(x)}$. Using $g(x)=x\ln x$ and $h(x)$, we obtain

$$e^{h(x)}=\left(1-\frac{\alpha^2x^2}{(x^2+1)^2}\right)^{x^2-\alpha x+1}\left(1+\frac{\alpha x}{x^2+1}\right)^{2\alpha x}.$$

From $\lim_{x\to\infty}\left(1+\frac{p}{x}\right)^x = e^p$, we obtain $\lim_{x\to\infty} e^{h(x)} = e^{\alpha^2}$, and thus, $\lim_{x\to\infty} h(x) = \alpha^2$.

Moreover, $h^{(3)}(x)$ can be computed as

$$h^{(3)}(x) = -\frac{4\alpha^2 x(x^2-1)\{\alpha^2(x^4+4x^2+1)-6(x^2+1)^2\}}{(x^2+1)^2(x^2-\alpha x+1)^2(x^2+\alpha x+1)^2}.$$

As $\alpha \geq \sqrt{6}$, we obtain $h^{(3)}(x) < 0$ in (α, ∞). Thus, we complete the proof. $\square$

We present the following theorem, which is the main theorem of this subsection.

Theorem 7. *The running time complexity of the proposed modified Shell sort algorithm is obtained as $O(n^{3/2}\sqrt{\alpha + \log\log n})$. For $\alpha \geq \sqrt{6}\log e - 1$, its SFP is upper-bounded by $2^{-\alpha}$.*

Proof. As the swapping operation in the modified Shell sort algorithm can be performed within a certain constant time, the running time complexity of the modified Shell sort in Algorithm 1 is determined from the number of swapping operations. Let $S(n)$ be the number of the swapping operations with an input size n. Then, $S(n)$ can be upper-bounded as

$$S(n) \leq n \sum_{\ell=0}^{\lfloor \log n \rfloor} k_\ell$$

where the window length k_ℓ of each gap is defined as

$$\left\lceil \sqrt{\left\lceil \frac{n}{2^{\ell+1}} \right\rceil \cdot (\alpha + 1 + \log\log n + \ell) \cdot \frac{1}{\log e}} \right\rceil.$$

Thus, $S(n)$ can be expressed as

$$S(n) = O\left(n^{\frac{3}{2}} \sum_{\ell=0}^{\lfloor \log n \rfloor} \sqrt{\frac{\alpha + \log\log n + \ell + 1}{2^{\ell+1}}} \right).$$

Using $\sqrt{a+b} \leq \sqrt{a} + \sqrt{b}$, we obtain

$$T(n) = O\left(n^{\frac{3}{2}} \left(\sqrt{\alpha + \log\log n} \sum_{\ell=1}^{\lfloor \log n \rfloor + 1} \frac{1}{2^{\frac{\ell}{2}}} + \sum_{\ell=1}^{\lfloor \log n \rfloor + 1} \frac{\sqrt{\ell}}{2^{\frac{\ell}{2}}} \right) \right).$$

Thus, we obtain $S(n) = O(n^{3/2}\sqrt{\alpha + \log\log n})$, because $\sum_{\ell=1}^{\infty} \frac{1}{2^{\frac{\ell}{2}}}$ and $\sum_{\ell=1}^{\infty} \frac{\sqrt{\ell}}{2^{\frac{\ell}{2}}}$ are both finite.

At this point, we consider the SFP. Let B denote the event that the output of the sorting algorithm is not successfully sorted and let B_ℓ denote the event that at least one subarray for the gap 2^ℓ is not successfully sorted. As $\mathsf{B} \subseteq \bigcup_{\ell=0}^{\lfloor \log n \rfloor} \mathsf{B}_\ell = \bigcup_{\ell=0}^{\lfloor \log n \rfloor} \left(\mathsf{B}_\ell \cap \bigcap_{u=\ell+1}^{\lfloor \log n \rfloor} \mathsf{B}_u^c \right)$, we obtain

$$\Pr[\mathsf{B}] \le \sum_{\ell=0}^{\lfloor \log n \rfloor} \Pr\left[\mathsf{B}_\ell \cap \bigcap_{u=\ell+1}^{\lfloor \log n \rfloor} \mathsf{B}_u^c \right] \le \sum_{\ell=0}^{\lfloor \log n \rfloor} \Pr\left[\mathsf{B}_\ell \middle| \bigcap_{u=\ell+1}^{\lfloor \log n \rfloor} \mathsf{B}_u^c \right]$$

where $\bigcap_{u=\ell+1}^{\lfloor \log n \rfloor} \mathsf{B}_u^c$ implies the event that the sorting is successful for the gaps $2^{\ell+1}, \cdots, 2^{\lfloor \log n \rfloor}$. All of the subarrays satisfy the condition $a_i < a_{i+2}$ in Theorem 4, before we perform the insertion sort for the gap 2^ℓ. Clearly, there are 2^ℓ subarrays when the gap is 2^ℓ, and the length of subarray is less than or equal to $2\lceil \frac{n}{2^{\ell+1}} \rceil$. Let $m_\ell = \lceil \frac{n}{2^{\ell+1}} \rceil$, and $\beta_\ell = \sqrt{(\alpha + 1 + \log\log n + \ell) \cdot \frac{1}{\log e}}$. As $\beta_\ell \ge \sqrt{6}$, the probability that one subarray of length $2m_\ell$ is not successfully sorted can be upper-bounded as

$$1 - \frac{C(2m_\ell, \beta_\ell \sqrt{m_\ell})}{\binom{2m_\ell}{m_\ell}} \le 2 \frac{\binom{2m_\ell}{m_\ell - \beta_\ell \sqrt{m_\ell}}}{\binom{2m_\ell}{m_\ell}} \le 2e^{-\beta_\ell^2}$$

where the second inequality is obtained from Lemma 6. We then obtain

$$\sum_{\ell=0}^{\lfloor \log n \rfloor} \Pr\left[\mathsf{B}_\ell \middle| \bigcap_{u=\ell+1}^{\lfloor \log n \rfloor} \mathsf{B}_u^c \right] \le \sum_{\ell=0}^{\lfloor \log n \rfloor} 2^\ell \cdot 2e^{-\beta_\ell^2} = \frac{2^{-\alpha} \lfloor \log n \rfloor}{\log n} \le 2^{-\alpha},$$

and thus, the theorem is proved. □

4 Optimal Window Length by Convex Optimization

It is necessary to find the shortest window length for the SFP so that the least running time complexity of the modified Shell sort is obtained. Generally, it is not easy to derive the optimal window length in closed form. In this section, we obtain the optimal window length using convex optimization. Let $\beta_\ell \sqrt{\lceil n/2^{\ell+1} \rceil}$ be the window length for the gap 2^ℓ, and $\Pr\left[\mathsf{B}_\ell \middle| \bigcap_{u=\ell+1}^{\lfloor \log n \rfloor} \mathsf{B}_u^c \right]$ be the SFP for the gap 2^ℓ, when sorting is successful for the gaps $2^{\ell+1}, \cdots, 2^{\lfloor \log n \rfloor}$. From Theorem 4 and Lemma 6, we obtain

$$\Pr\left[\mathsf{B}_\ell \middle| \bigcap_{u=\ell+1}^{\lfloor \log n \rfloor} \mathsf{B}_u^c \right] \le 2^\ell e^{-\beta_\ell^2}.$$

The objective function that needs to be minimized is the total number of swap operations, which determines the running time. As the exact running time formula is rather complicated, we consider a tight upper bound of the running time, $n \sum_{\ell=0}^{\lfloor \log n \rfloor} \beta_\ell \sqrt{\lceil n/2^{\ell+1} \rceil}$, which is used in the proof of Theorem 7. Let

$p_\ell = 2^\ell e^{-\beta_\ell^2}$. Then, we have $\beta_\ell = \sqrt{(\ell + \log(1/p_\ell))/\log e}$. As it is sufficient to minimize $\sum_{\ell=0}^{\lfloor \log n \rfloor} \sqrt{\lceil n/2^{\ell+1} \rceil (\ell + \log(1/p_\ell))}$, the problem of the optimal window length can be formulated as follows;

$$\begin{aligned} \textbf{minimize} \quad & \sum_{\ell=0}^{\lfloor \log n \rfloor} \sqrt{\lceil n/2^{\ell+1} \rceil (\ell + \log(1/p_\ell))} \\ \textbf{s.t.} \quad & \sum_{\ell=1}^{k-1} p_\ell \le p_{\text{err}}. \end{aligned}$$

This formulation implies that the total running time with SFP upper-bounded by p_{err} needs to be minimized. We can validate that $\sqrt{c + \log \frac{1}{x}}$ is a convex function on small positive values, where c is a constant. As the weighted sum of convex functions is also a convex function, the objective function is a convex function, and the constraint is also convex. Thus, this can be termed as a convex optimization problem. As every convex optimization problem can be solved using numerical analysis, it is easy to obtain the optimal window length. Then, we can deduce p_ℓ, and the optimal window length is determined to be $\lceil \sqrt{\lceil n/2^{\ell+1} \rceil (\ell + \log(1/p_\ell))/\log e} \rceil$ for each gap 2^ℓ. It is noted that the above formulation is not sufficiently tight, because it still uses the union bound. Constructing a tighter formulation, which can be solved easily, can be a focus for future research.

5 Simulation Results

The performance of the proposed modified Shell sort is numerically verified using a personal computer with an AMD Ryzen 7 1700 CPU running at 3GHz, and 16GB RAM. First, we validate the running time and SFP when the array size varies. Then, the running time and SFP are numerically obtained when the parameter α is varied. Finally, the performance of the modified Shell sort is compared with the cases corresponding to the optimal window length, which is obtained using convex optimization, and Ciura's optimal gap sequence, which has been validated numerically as an optimal gap sequence in non-FHE settings.

The running time is mainly determined by the product of the number of swapping operations and the running time of the maximum or minimum function. Clearly, the running time of the maximum or minimum function is independent of the input array size or α. The relation between the performance and the main parameters of the proposed modified Shell sort is not significantly affected by the use of the homomorphic encryption scheme. Thus, in our numerical analysis, we do not use the actual homomorphic encryption scheme.

Fig. 4 shows the relation between the running time and SFP against various array sizes for $\alpha = 3$. It is observed that the array size increases from 50 to 1000. The input arrays are randomly generated, and 10^5 input arrays are generated for each array size. It is observed from Fig. 4 that the running time increases in proportion to $n^{3/2}$, and the SFP is independent of the array size. This numerical result coincides well with the proposed analysis of the modified Shell sort.

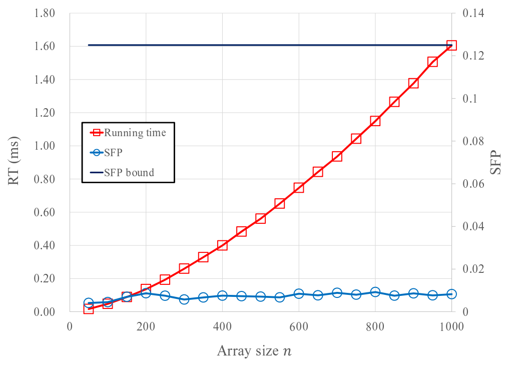


Fig. 4: Running time and SFP of the modified Shell sort for varied array sizes.

Fig. 5 shows the relation between the running time and SFP for various α values, in which g2p denotes the power of the 2-gap sequence, gop denotes Ciura's optimal gap sequence, and a-win and o-win denote the analytically derived window length and optimal window length derived by convex optimization, respectively. The input array size is fixed at 1000. Similar to the previous simulation, 10^5 input arrays are randomly generated for each α value. Algorithm 1 and the case corresponding to the Ciura's optimal gap sequence or optimal window length are simulated, with the optimal window length derived using the convex optimization discussed in Section 4.

From Fig. 5, it is observed that the running time of Algorithm 1 increases as α increases and the growth rate decreases. This observation coincides with the proposed analysis, i.e., the running time is approximately proportional to $\sqrt{\alpha}$. The logarithms of the SFP values of Algorithm 1 are parallel to that of the SFP bounds. This implies that the SFP is proportional to $2^{-\alpha}$ with some small proportional constant.

When the gap sequence is replaced with Ciura's gap sequence, the running time is reduced by approximately 0.5 ms. Sorting failure is not detected in the case of the simulation that uses Ciura's gap sequence. This implies that the order of the SFP of Ciura's optimal gap sequence is less than or equal to 10^{-5}. Although the window lengths of each gap in this paper are analytically derived for the power of the 2-gap sequence, a better result is obtained when Ciura's optimal gap sequence is used. Further analyses on using Ciura's optimal gap sequence will be included in future studies.

The optimal window length is derived using the convex optimization problem described in Section 4. The running time in this case is marginally reduced compared with the case using the analytically obtained window length. However, their values become closer as α increases. The SFP of the case using the optimal window length for the power of the 2-gap sequence is closer to the SFP bound than that of the case using the analytically obtained window length. Thus, the running time can be reduced, while the SFP remains less than the SFP bound.

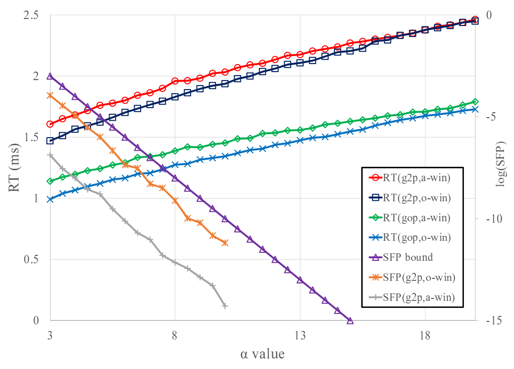


Fig. 5: Running time and SFP of the modified Shell sort for varied α values and comparison of these values with those obtained from the cases of Ciura's optimal gap sequence and optimal window length derived by convex optimization.

6 Conclusion and Future Work

In this paper, we proposed a modified Shell sort with a gap sequence of powers of two and an additional parameter α in the FHE setting, and derived the running time complexity $O(n^{3/2}\sqrt{\alpha + \log\log n})$, considering a trade-off with the SFP $2^{-\alpha}$. We also established that the running time complexity of the proposed algorithm is almost the same as the average-case running time complexity of the original Shell sort, while the SFP is maintained to be minimal. We then obtained the optimal window length of each gap by numerically solving a convex optimization problem. We believe that this study plays a significant role in the foundation of the analysis of the Shell sort in FHE settings.

We plan to use the Shell sort with other gap sequences in a future study by extending this analysis.

References

1. Chatterjee, A., Kaushal, M., Sengupta, I.: Accelerating sorting of fully homomorphic encrypted data. In: International Conference on Cryptology in India. pp. 262–273. Springer

2. Chatterjee, A., Sengupta, I.: Windowing technique for lazy sorting of encrypted data. In: 2015 IEEE conference on communications and network security (CNS). pp. 633–637. IEEE (2015)
3. Chatterjee, A., SenGupta, I.: Sorting of fully homomorphic encrypted cloud data: Can partitioning be effective? IEEE Transactions on Services Computing (2017)
4. Cheon, J.H., Kim, D., Kim, D., Lee, H.H., Lee, K.: Numerical methods for comparison on homomorphically encrypted numbers. IACR Cryptology ePrint Archive (2019)
5. Ciura, M.: Best increments for the average case of shellsort. In: International Symposium on Fundamentals of Computation Theory. pp. 106–117. Springer (2001)
6. Emmadi, N., Gauravaram, P., Narumanchi, H., Syed, H.: Updates on sorting of fully homomorphic encrypted data. In: 2015 International Conference on Cloud Computing Research and Innovation (ICCCRI). pp. 19–24. IEEE
7. Espelid, T.O.: Analysis of a shellsort algorithm. BIT Numerical Mathematics **13**(4), 394–400 (1973)
8. Gentry, C.: Fully homomorphic encryption using ideal lattices. In: Stoc. vol. 9, pp. 169–178
9. Knuth, D.E.: The art of computer programming: sorting and searching, vol. 3. Pearson Education (1997)
10. Rivest, R.L., Adleman, L., Dertouzos, M.L., et al.: On data banks and privacy homomorphisms. Foundations of secure computation **4**(11), 169–180 (1978)
11. Shell, D.L.: A high-speed sorting procedure. Communications of the ACM **2**(7), 30–32 (1959)