AIMS Mathematics

AIMS Mathematics, 9(5): 11024–11038.
DOI: 10.3934/math.2024540
Received: 20 January 2024
Revised: 03 March 2024
Accepted: 11 March 2024
Published: 20 March 2024

http://www.aimspress.com/journal/Math

Research article

An efficient simulated annealing algorithm for short addition sequences

Hazem M. Bahig[1,*], Mohamed A.G. Hazber[1] and Hatem M. Bahig[2]

[1] Department of Information and Computer Science, College of Computer Science and Engineering, University of Ha'il, Ha'il 81481, KSA
[2] Department of Mathematics, Faculty of Science, Ain Shams University, Cairo, Egypt

* **Correspondence:** Email: h.bahig@uoh.edu.sa.

Abstract: Let $N = \{n_1, n_2, \ldots, n_k\}$ be a finite set of positive numbers. The problem of finding the minimal number of additions required to compute all elements of *N* starting from 1 (called the addition sequence problem) is NP-complete. It is equivalent to finding the minimum number of multiplications needed to compute a group exponentiation g^{n_1}, $g^{n_2}, \ldots, g^{n_k}$, where g is an element in a group. This paper aims to propose a new metaheuristic algorithm using a simulated annealing strategy to generate a short addition sequence. The performance of the proposed algorithm is measured by considering two parameters: The size of *N* and the domain of $n_i, 1 \le i \le k$. The proposed algorithm is a new trade-off between the length of the generated addition sequence and the average running time of generating addition sequences. It sometimes produces longer addition sequences than exact algorithms that are slower, and it is slower than suboptimal algorithms that produce longer addition sequences.

Keywords: addition sequence; addition chain; modular exponentiation; simulated annealing; heuristic algorithm; optimization
Mathematics Subject Classification: 11Bxx, 90C59, 68T20

1. Introduction

Given a set of numbers $N = \{n_1, n_2, \ldots, n_k\}$ such that $1 < n_1 < n_2 < \cdots < n_k$. An addition sequence [1,2] for the set *N*, denoted by *ASeq*(*N*), is an increasing sequence of numbers $as_0, as_1, as_2, \ldots, as_l$ such that

(1) $as_0 = 1$,
(2) $as_l = n_k$,

(3) $as_i = as_j + as_h, 0 \le j, h \le i-1,$

(4) $N \subseteq \{as_0, as_1, as_2, \ldots, as_l\}$, i.e., each number n_i should appear in the sequence $as_0, as_1, as_2, \ldots, as_l$.

The length of *ASeq*(*N*) is equal to *l*, and the minimal value of *l* is denoted by $\ell(N)$. In the case of $k = 1$, the sequence is called addition chain [1,2]. One of the important aspects of generating the shortest *ASeq(N)* is that it is equivalent to the simultaneous evaluation of *k* power monomials

$$g^{n_1}, g^{n_2}, \ldots, g^{n_k}$$

with a minimum number of multiplications.

For example, let $N = \{53, 163, 203, 363\}$. The following are two *ASeq*s with lengths of 15 and 13, respectively. The first *ASeq* is: 1, 2=1+1, 3=2+1, 6=3+3, 12=6+6, 13=12+1, 26=13+13, 39=26+13, 40=39+1, 53=40+13, 106=53+53, 159=106+53, 160=159+1, 163=160+3, 203=163+40, 363=203+160.

The second *ASeq* is: 1, 2=1+1, 3=2+1, 5=3+2, 10=5+5, 13=10+3, 20=10+10, 40=20+20, 53=40+13, 80=40+40, 160=80+80, 163=160+3, 203=163+40, 363=203+160.

The computation of $g^{53}, g^{163}, g^{203}, g^{363}$, using the first sequence is

$$g, g^2, g^3, g^6, g^{12}, g^{13}, g^{26}, g^{39}, g^{40}, g^{53}, g^{106}, g^{159}, g^{160}, g^{163}, g^{203}, g^{363},$$

while the computation of the same powers $g^{53}, g^{163}, g^{203}, g^{363}$ using the second sequence is

$$g, g^2, g^3, g^5, g^{10}, g^{13}, g^{20}, g^{40}, g^{53}, g^{80}, g^{160}, g^{163}, g^{203}, g^{363}.$$

A step *i* is called

(1) star if $as_i = as_{i-1} + as_h, 0 \le h \le i-1$; and

(2) non-star if $as_i = as_j + as_h, 0 \le j, h \le i-2$.

In case, $j = h = i-1$, $as_i = 2as_{i-1}$, the step is called doubling. If all steps in the sequence are stars, then the sequence is called a star. If $\ell^*(N)$ denotes the minimal length of star *ASeq* (*N*), then we have

$$\ell(N) \le \ell^*(N) \tag{1}$$

Yao [3] showed that:

$$\ell(N) \le \log n_k + (c\,k) \log n_k / \log\log n_k, \tag{2}$$

where $c = 2 + 4/\sqrt{\log n_k}$.

Bleichenbacher [4] computed the lower bound

$$\ell(N \cup \{n_{k+1}\}) \ge \ell(N) + \alpha + 1, \tag{3}$$

where $n_{k+1} > 2^\alpha n_k, \alpha \ge 0$.

ASeqs have received a lot of consideration among mathematicians and computer scientists for the following reasons. The first reason is that one of the fundamental operations, that play a crucial function in the efficiency of many public key cryptosystems and protocols, such as RSA [5], is group exponentiation (sometimes it is called *multi-modular exponentiation* [1]), *i.e.*, computing g^{n_1}, $g^{n_2}, \ldots, g^{n_k}$ simultaneously with a minimal number of operations, where *g* is an element in a group. Designing a fast algorithm for generating a shortest (or short) *ASeq* increases the efficiency of such public key cryptosystems and protocols.

The second reason is that *ASeq*s (including addition chains) are generalized to the following:

- B-chains [6], where every element in the B-chain has the form $as_i = as_j \; o \; as_h, 0 \leq j, h \leq i-1,$ and the binary operation o belongs to a finite set of binary operations B over the set of natural numbers, i.e. $o \in B = \{+, -, *, \div\}$. Guzmán-Trampe et. al. [7] proposed a method for generating addition-subtraction (*i.e.,* $B = \{+, -\}$) sequence for the Kachisa–Schaefer–Scott family of pairing-friendly elliptic curves.
- Vectorial addition chain [8, 9]: it is a sequence of k-dimensional vectors of nonnegative integers $v_i, -k+1 \leq i \leq l$, such that (1) $v_{-k+1} = [1,0,0 \dots ,0,0]$, $v_{-k+2} = [0,1,0 \dots ,0,0]$, …, $v_0 = [0,0,0 \dots ,0,1]$; (2) $v_i = v_j + v_h, 1 \leq i \leq l, -k+1 \leq j, h \leq i-1$; (3) $v_l = [n_1, n_2, \dots, n_k]$. Finding a shortest vectorial addition chain is equivalent to evaluating multi-exponentiation, i.e., the product $\prod_{i=1}^{k} g_i^{n_i}$ with the minimal number of multiplications.

The third reason for the importance of ASeq is that in Internet of Things, IoT, devices with limited resources have a problem when they perform some public-key primitives, such as decryption and signature which involve modular exponentiation, because most public-key primitives are (*i*) time-consuming compared with symmetric-key cryptosystems; and (*ii*) using private information. One of the common solutions to this problem is to use what is called "server aided secret computation protocols," denoted by SASCP [10–12], or sometimes it is called outsourcing protocols [13]. In such protocols, devices with limited power and resources can execute public-key primitives efficiently with the aid of an untrusted powerful server without revealing private information. Another and similar solution to the problem is to define a delegation protocol [14–16]. This is a protocol that satisfies two security requirements: (*i*) Privacy, which prevents passive attackers from recovering private information, and (*ii*) verifiability, which prevents an untrusted server from forcing the devices to accept a false value as the outcome of the delegated computation.

The main challenge of finding a shortest ASeq is that it is NP-complete [9]. Additionally, when the size of *N* is large and the size of exponents is large, the running time for finding a shortest ASeq is very large. On the other side, designing a suboptimal algorithm generates ASeq with a long length. Therefore, designing a fast algorithm for generating a short (not necessarily shortest) ASeq is interesting using metaheuristics techniques, especially a simulated annealing strategy.

A simulated annealing (SA) algorithm [17] is an iterative metaheuristic algorithm used to solve optimization problems. It was proposed as an adaptation of the Metropolis method to simulate the thermal moves of molecules at a fixed temperature *T*.

A SA algorithm starts with the initial state (solution) S and sets the current state to *S*. Then, it randomly generates a new state S' from the neighbors of the current state *S* and decides whether to accept S' or not based on the probability function. If S' is accepted, then *S* is replaced by S'. Otherwise, a random real number α is generated and if α is less than the value of the probability function then the generated state S' is accepted. These steps are repeated based on the number of metropolis cycles *m* for a fixed temperature. Then, the algorithm updates the temperature and repeats this process until it reaches the maximum number of annealing iterations.

In this paper, we propose a new metaheuristic algorithm based on a simulated annealing strategy to find a short ASeq for the set *N*. The proposed simulated annealing algorithm generates an addition sequence with lengths shorter than the lengths generated by the previous suboptimal algorithms. It also shows that the execution time of the proposed algorithm is significantly faster than the exact algorithm that generates shortest addition sequences. Thus, the proposed algorithm is a new trade-off between the size of the generated ASeq and the average running time of generating ASeq.

The organization of the remainder of this paper is as follows. Section 2 includes the previous works of ASeq. In Section 3, the outlines and details of the proposed algorithm are given. Section 4 describes the implementation of three algorithms (the exact, a fast suboptimal, and the proposed algorithms). This section includes the dataset used in the experiments and an analysis of the experimental results for the three algorithms. Finally, Section 5 includes the conclusion of this paper and future works.

2. Related work

ASeq Algorithms are divided into two categories. The first category is to find a shortest ASeq. In fact, there are a few papers that discuss a generation of shortest ASeqs. Bleichenbacher [4] suggested an algorithm to find a shortest *ASeq*(*N*) with length *r* provided that the algorithm previously computed $\ell(y)$ for all numbers $y < maximum(N)$, and $\ell(y) < r$. The author used the suggested algorithm to find a shortest addition chain up to a certain number.

The authors in [18] generated a shortest *ASeq*(*N*) based on a branch and bound search algorithm. The algorithm begins by calculating a lower bound, Eq (3), and looking for an addition chain for the first element n_1 in the set *N*. Then, it extends the chain to an addition sequence for $\{n_1, n_2\}$, and so on until it generates *ASeq*(*N*). The algorithm uses different strategies to speed up the generation as follows. (*i*) Using bounding sequences to prune some branches in the search tree which cannot lead to a shortest ASeq. (*ii*) Determining an upper bound of $\ell(\{n_1, n_2, \dots, n_i\}), 1 \leq i \leq k.$ (*iii*) Using some sufficient conditions for star steps to skip the generation of non-star steps. *(vi)* If no *ASeq*(*N*) of length *l* is found, then the algorithm increases *l* by one and repeats the process until either *l* is equal to the length of the generated short ASeq produced by continued fraction (CF) method [19] or the algorithm finds a shortest ASeq. Recently, the authors in [20,21] used multicore systems to improve the generation of a shortest ASeq.

The second category is to find a short ASeq. Yao [3] presented an algorithm to compute g^{n_1}, $g^{n_2}, \dots, g^{n_k}$ in $O(\lg n_k + c\sum_{i=1}^{k}(\log n_i/loglog(n_i+2))$ multiplications for some constant *c*. Bos and Coster [22] proposed four methods to generate a short ASeq and used them in the window method [2,22]. Experimentally, for $n_k \leq 1000,$ the estimated upper bound of the lengths for the generated ASeqs by the four methods is

$$\ell(\{n_1, n_2, \dots, n_k\}) \leq \frac{3}{2}\log n_k + k + 1, \tag{4}$$

Bergeron et al. [19] proposed an efficient method based on CF. The suggested method can be considered an extension and unifying approach of some previously known methods (such as binary and *k*-ary methods [1,2]) for generating a short addition chain, i.e., ASeq with *k*=1. Recall that the CF expansion of n/d is

$$\frac{n}{d} = c_t + \cfrac{1}{c_{t-1} + \cfrac{1}{\ddots + \cfrac{1}{c_2 + \cfrac{1}{c_1}}}} \tag{5}$$

where *d* is an integer in the interval $[2, n-1]$.

Bergeron et al. [19] suggested different strategies for choosing the value of *d.* One of the efficient strategies that produces a good suboptimal ASeq is the dichotomic strategy, where

$$d = \left\lfloor \frac{n}{2^{\lceil \lfloor log_2 n \rfloor / 2 \rceil}} \right\rfloor \tag{6}$$

Let $N' = \{n_k, n_{k-1}, \dots, n_1\}$, and $\mathcal{L}(N')$ denotes the length of the $ASeq(N')$ generated by CF using the dichotomic strategy. Then

$$\mathcal{L}(N') = \begin{cases} \mathcal{L}(N' \setminus \{n_k\}) + \ell(q), & if\ rem = 0; \\ \mathcal{L}(N' \setminus \{n_k\}) + \ell(q) + 1, & if\ rem = 1{,}2 \\ \mathcal{L}(N' \cup \{rem\} \setminus \{n_k\}) + \ell(q) + 1, & \text{otherwise} \end{cases} \tag{7}$$

where $n_k = q\, n_{k-1} + rem,\ rem < n_{k-1}$ and

$$\ell(n) = \begin{cases} \alpha, & if\ n = 2^{\alpha}; \\ 3, & if\ n = 3; \\ \mathcal{L}(\{n, d\}), & \text{otherwise} \end{cases} \tag{8}$$

where *d* is defined by Eq (6).

Enge et. al. [23] proposed a special method to construct a short ASeq to find the first *k* nonzero terms in the sparse q-series belonging to the Dedekind eta function or the Jacobi theta constants. Nadia and Mourelle [24] used the Ant Colony strategy to find a short ASeq. They tested the strategy on a small set of numbers. Abbas and Gustafsson [25] proposed a method based on integer linear programming to generate a short ASeq for a small set of numbers.

However, an extensive study is lacking to identify the difference in length between (1) the exact algorithm that generates the shortest ASeqs; and (2) suboptimal algorithms that generate short ASeqs. Two important parameters should be considered in the implementation. The first one is the domain of the numbers $n_i, 1 \leq i \leq k.$ The other is the cardinality of *N*. In addition to the length, we compare the running times of algorithms. Another lacking direction of research is how to use the SA strategy to find a short ASeq.

3. The proposed method

This section presents a description of the proposed Simulated Annealing algorithm for Addition Sequence, denoted by SAAS.

Initially, the algorithm starts by generating the initial state, AS_0, using the CF method [19], and its energy is equal to the length l_{AS_0} of AS_0. Then, the algorithm assigns these two values to the best state and the best energy, respectively. After that, the algorithm repeats the following steps based on the number of Metropolis cycles, *metropolisNo*, for a fixed temperature. In each iteration of this loop, the algorithm performs the following steps:

The first step is generating a new state, AS_{new}, and its energy, l_{new}. The second step is determining whether the algorithm accepts this new state or not. The algorithm accepts the new state and its energy, and then assigns these values to the best state and best energy if either of the following conditions is true. (1) If the energy of the new state is lower than the energy of the best state. (2) If the Boltzmann distribution is greater than a random real number in the range [0,1].

After completing the number of Metropolis cycles for a fixed temperature, the algorithm updates the temperature using the Kirkpatrick quenching method and repeats this process until it reaches the maximum number of annealing iterations.

The details of the algorithm steps are as follows.

Step 1: Generate the initial state, AS, using the CF method for the set of exponents $N = \{n_1, n_2, \dots, n_k\}$, where $AS = \{as_0, as_1, as_2, \dots, as_l\}$ such that (1) $as_0 = 1$ and $as_1 = 2$; (2) $\exists\, i, l_i$ s.t. $n_i = as_{l_i}$ and $1 \le i \le k$; (3) $L = \{l_1, l_2, \dots, l_k\}$ such that $l_i < l_{i+1}$ and $l = l_k$.

Step 2: Repeat the following *metropolisNo* times:

Step 2.1: Generate a random integer number, *r*, from the interval $[0, k-1]$. This number will be used as a starting point for new neighbor based on the elements of *N*.

Step 2.2: Generate a new state, AS_{new} from the location l_r. If *r*=0, then the algorithm finds the new state from the element $as_1 = 2$ of *AS*. Otherwise, the algorithm finds the new state from $as_{l_r} = n_r$ to n_k. The process of generating the new elements from n_i to n_{i+1} is based on the following rules.

- Rule # 1: Doubling the current element, i.e., $as_{j+1} = 2\,as_j$.
- Rule # 2: Summing the last two elements, i.e., $as_{j+1} = as_j + as_{j-1}$.
- Rule # 3: Summing the last element with any other random element in the sequence, $as_{j+1} = as_j + as_h,\ 0 \le h < j$.

This step can be done as follows (Steps 2.2.1–2.2.3).

Step 2.2.1 (Generate one element in the sequence): If the current goal is n_{i+1} and the current *ASeq* is $\{as_0, as_1, \dots, as_{l_i} = n_i, as_{l_i+1}, \dots, as_{l_i+j}\}, j \ge 0$, then the steps of generating a new element in the chain are as follows.

```
1.  $d = n_{i+1} - as_{l_i+j}$
2.  If $d = as_{l_i+j}$ then apply Rule # 1
3.  Else if $d = as_{l_i+j-1}$ then apply Rule # 2
4.      Else if $d > as_{l_i+j}$ then
5.              Generate a random real number $\alpha \in [0,1]$
6.              If $\alpha \ge 0.5$ then apply Rule # 1
7.              Else
8.                  Generate a random real number $\alpha \in [0,1]$
9.                  If $\alpha \ge 0.5$ then apply Rule # 2
10.                 Else
11.                     Generate a random integer number $r \in [0, l_i + j - 2]$
12.                     Apply rule # 3, where $h=r$.
13.       Else // $d < as_{l_i+j}$
14.           Generate a random integer number $r \in [0, l_i + j - 2]$
15.           Apply Rule # 3, where $h=r$.
16.           If the new element is less than or equal to $n_{i+1}$ then the element is accepted.
                Otherwise, decrease the value of r and apply rule #3 until the algorithm finds a
                certain value of h such that the new element is less than or equal to $n_{i+1}$.
```

Step 2.2.2 (Generate all elements between n_i and n_{i+1}): Repeat Step 2.2.1 starting from $j = 0$, and $as_{l_i} = n_i$, until the algorithm finds $as_{l_i+j_i} = n_{i+1}$. In this case, the algorithm updates the value of $l_{i+1} = l_i + j_i, 1 \le j_i$.

Step 2.2.3 (Generate the ASeq from n_r to n_k): Repeat Steps 2.2.1 and 2.2.2 until the algorithm generates n_k. Therefore,

$$AS_{new} = \{as_0, as_1, \dots, n_r = as_{l_r}, as'_{l_r+1}, \dots, n_{i+1} = as'_{l_i+j_i}, \dots, n_k = as'_{l_{k-1}+j_{k-1}} = as'_{j_{k-1}}\};$$

$$L_{new} = \{l_1, l_2, \dots, l_r, l'_{r+1}, l'_{r+2}, \dots, l'_k\}.$$

Step 3: Test the acceptance of the new state by the following steps.

```
1.    If $l'_k < l$ then $AS = AS_{new}$ and $l = l'_k$
2.    Else generate a random real number $\alpha$.
3.        $d_e = l'_k - l$
4.        If $e^{-d_e/T} > \alpha$ then $AS = AS_{new}$ and $l = l'_k$
```

Step 4: Decrease the temperature using Kirkpatrick quenching method: $T = \gamma T$, where $\gamma = 0.99$.

Step 5: Repeat Steps 2, 3, and 4 until the algorithm reaches the maximum number of annealing iterations.

Remarks:

(1) Each random number is chosen uniformly.

(2) The two symbols r, r' are used for integer random numbers, while the symbol α is used for a real random number.

(3) The time complexity of the proposed algorithm is O (*SuccNo* * *metropolisNo* * *log* n_k), where the term *log* n_k represents the running time to generate one ASeq, see Eq (2).

(4) Figure 1 shows how to generate a random ASeq by applying the three rules for {13, 85, 254}.

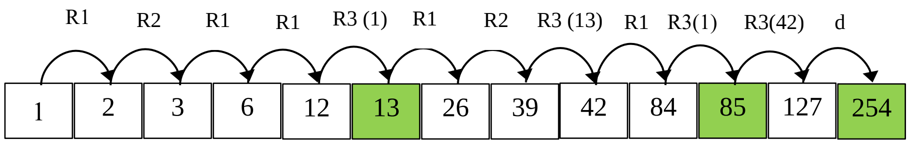


Figure 1. Example of randomly generating a short ASeq for the set {13, 85, 254}.

Algorithm 1 (SAAS) and Algorithm 2 (ASNeighbours) represent the complete pseudocodes for the proposed method. The SAAS algorithm starts with assigning the initial value of temperature to *T* and then generates an initial ASeq using the CFAS method. After that the SAAS algorithm generates randomly a new state by calling ASNeighbours algorithm and decides whether to accept this new state or not. The ASNeighbours algorithm assigns the ASeq for $\{n_1, n_2, \dots, n_r\}$ to the new state and then generates new elements for the remainder set $\{n_{r+1}, n_{r+2}, \dots, n_k\}$.

Algorithm 1: SAAS

```
Input: $N = \{n_1, n_2, \dots, n_k\}$, // a finite set of positive numbers
       $T_0$ // initial temperature
       $\gamma$ // cooling factor
       succNo // the maximum number of annealing iterations
       metropolisNo // number of Metropolis cycles
Output: $AS = \{as_0, as_1, \dots, as_{l_k}\}$
1.  $T = T_0$
2.  $CFAS(N, AS, L, l)$                    // $L = \{l_1, \dots, l_r, \dots l_k\}, l = l_k$.
3.  for $suc = 0$ to $succNo$ -1 do
4.      for $m = 1$ to metropolisNo do
5.          Generate a random integer number $r \in [0, k-1]$
6.          $ASNeighbours(AS, L, r, AS_{new}, L_{new})$
7.          $d_l = l_{new} - l$
8.          if $d_l < 0$ then
```

```
9.                $AS = AS_{new}$
10.               $L = L_{new}$
11.               $l = l_{new}$
12.           else
13.               Generate a random real number $\alpha \in [0,1]$
14.               if $e^{-d_l/T} > \alpha$ then
15.                   $AS = AS_{new}$
16.                   $l = l_{new}$
17.                   $L = L_{new}$
18.       $T = \gamma T$
```

Algorithm 2: ***ASNeighbours***

```
Input: $AS = \{as_0, as_1, \dots, as_{l_k}\}$ // current ASeq
       $L = \{l_1, \dots, l_r, \dots l_k\}$ // energy of ASeq, i.e., $l_i$ is the index of $n_i$ in AS
       $r$ // start position to generate a new neighbor; $1 \le r < k$.
Output: $AS_{new} = \{as'_0, as'_1, \dots, as'_{l'_k}\}$ , $L_{new} = \{l'_1, l'_2, \dots, l'_k\}$.
1.  for $i = 1$ to $r$ do
2.    $l'_i = l_i$
3.  for $i = 0$ to $l_r$ do
4.    $as'_i = as_i$
5.  for $i = r$ to $k - 1$ do          // from $n_i$ we generate $n_{i+1}$
6.     $j = 0$
7.     while $as'_{l_i+j} \neq as_{l_{i+1}}$ $(= N_{i+1})$ do
8.        $d = as_{l_{i+1}} - as'_{l_i+j}$
9.        if $d = as'_{l_i+j}$ then $as'_{l_i+j+1} = 2\,as'_{l_i+j}$ , $j = j + 1$
10.       else if $d = as'_{l_i+j-1}$ then $as'_{l_i+j+1} = as'_{l_i+j} - as'_{l_i+j-1}$, $j = j + 1$
11.            else if $d > as'_{l_i+j}$ then
12.                 Generate a random real number $\alpha \in [0,1]$.
13.                 if $\alpha \ge 0.5$ then $as'_{l_i+j+1} = 2\,as'_{l_i+j}$ , $j = j + 1$
14.                 else
15.                     Generate a random real number $\alpha \in [0,1]$
16.                     if $\alpha \ge 0.5$ then $as'_{l_i+j+1} = as'_{l_i+j} + as'_{l_i+j-1}$, $j = j + 1$
17.                     else
18.                         Generate a random integer number $r' \in [0, l_i + j - 2]$ .
19.                         $as'_{l_i+j+1} = as'_{l_i+j} + as'_{r'}$ , $j = j + 1$
20.            else //$d_{as} < as_{l_i+j}$
21.                Generate a random integer number b $r' \in [0, l_i + j - 2]$ .
22.                $as'_{l_i+j+1} = as'_{l_i+j} + as'_{r'}$ , $j = j + 1$
23.                while $as'_{l_i+j} > as_{l_{i+1}}$ do
24.                    $r' = r' - 1$
25.                    $as'_{l_i+j} = as'_{l_i+j-1} + as'_{r'}$
26.    $l'_{i+1} = i + j$
```

4. Results and discussions

This section demonstrates the experimental study and its analysis for measuring the performance of the SAAS algorithm compared to the exact and suboptimal algorithms, ExAS and CFAS, respectively.

The three algorithms were programmed using the C language and run on a machine with a 2.5 GHz processor and a memory of 16 GB. Also, the three algorithms were compared by measuring the execution time in milliseconds and the length of the short/shortest sequence. The section consists of two subsections: Data generation and results.

4.1. Data generation

The data used in the experimental study is based on two factors. The first factor is the number of elements k in the set of exponents N. The experimental values of k are 2, 4, 6, 8, and 10. The second factor is the domain of each exponent in the set N. According to the window method and its variations [2,22], the range of exponents is the integer interval $[1, 2^e - 1]$, where e is the window length (of size e-bits). Also, according to the performance of the window method, the value of each exponent should be odd. The experimental values of e are equal to 7, 8, 9, and 10. The reason for starting the values of e with 7, the running times for all compared algorithms are fast when $e < 7$.

The methodology of generating the dataset is based on fixing the size of the window, i.e., e-bits, say e=7, and then generating different sets $N_{k,e}$ with lengths k= 2, 4, 6, 8, and 10. For each value of k, 25 sets of exponents in the range $[1, 2^e - 1]$ are generated. The process of generating different sets of exponents is as follows.

1. Set e to the maximum number of bits in the exponents, i.e., the window size.
2. Set the set $N_{0,e} = \emptyset$ and i=2.
3. While $i \leq k = 10$ do the following
   Construct a new set $N_{i,e}$, by adding two randomly generated odd numbers, in the range $[1, 2^e - 1]$, to the set $N_{i-2,e}, i.e.,$ $N_{i,e} = N_{i-2,e} \cup$ {the two generated randomly odd numbers}.
4. Set i=i+2.
5. Make sure that $N_{i,e}$ is sorted.
6. Repeat Steps 2-4, 25 times to generate 25 sets of exponents with at most e-bits.
7. Repeat Steps 1–5 for different sizes of exponents e=7, 8, 9, and 10.

The following example illustrates the generation of five sets with different values of k and a fixed size of exponent e=8.

$N_{2,8} = \{177, 241\}$.
$N_{4,8} = \{65, 125, 177, 241\}$.
$N_{6,8} = \{65, 89, 125, 177, 189, 241\}$.
$N_{8,8} = \{43, 65, 89, 125, 177, 189, 221, 241\}$.
$N_{10,8} = \{43, 65, 89, 103, 125, 177, 189, 203, 221, 241\}$.

The initial temperature, T_0, is equal to the number of instances used, which is equal to 25. The value of $\gamma = 0.99$.

4.2. Results

The results of executing the three algorithms on the generated data in terms of the length of the output are shown in Table 1. The first two columns represent the two factors *e* and *k*, while the three last columns represent the percentage of differences in the lengths of the output for the following cases: (1) ExAS and SAAS algorithms, (2) ExAS and CFAS algorithms, and (3) SAAS and CFAS algorithms. Since the exact algorithm always produces the shortest ASeq, the methodology of analyzing the results is to compute the number of instances in which the lengths of ASeqs generated by the SAAS and CFAS algorithms are longer than the lengths of shortest ASeqs generated by the ExAs algorithm. The percentages of these instances represent the third and fourth columns. Also, Table 1 presents the difference between the output of the SAAS and CFAS algorithms, see the last column in Table 1.

Table 1. Comparison between three algorithms in terms of the length of *ASeq*.

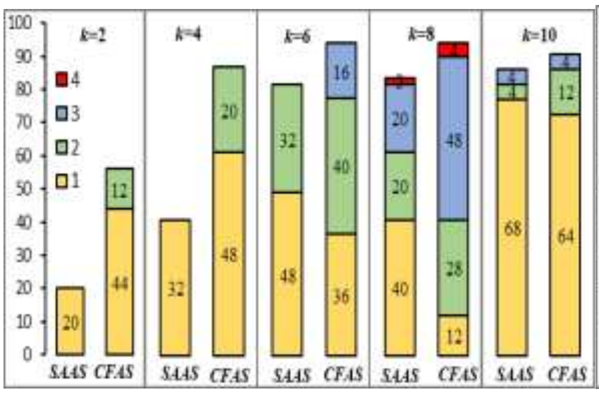
		Percentage of cases when		
e	k	$\lvert AS_{SAAS}\rvert > \lvert AS_{ExAS}\rvert$	$\lvert AS_{CFAS}\rvert > \lvert AS_{ExAS}\rvert$	$\lvert AS_{CFAS}\rvert > \lvert AS_{SAAS}\rvert$
7	2	12%	28%	16%
	4	40%	64%	36%
	6	56%	76%	44%
	8	76%	84%	20%
	10	88%	92%	20%
8	2	20%	56%	44%
	4	32%	68%	52%
	6	80%	92%	40%
	8	82%	92%	36%
	10	92%	96%	32%
9	2	16%	56%	44%
	4	44%	80%	56%
	6	84%	88%	36%
	8	92%	96%	32%
	10	96%	100%	28%
10	2	16%	72%	60%
	4	52%	80%	32%
	6	88%	100%	24%
	8	92%	100%	16%
	10	100%	100%	16%

The analysis of the data shows the following observations.

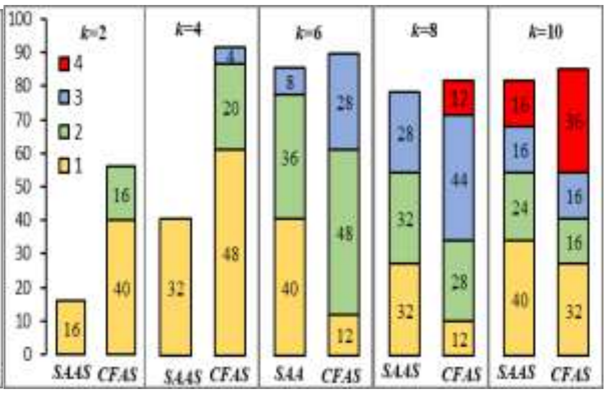
First, as shown in Table 1, the percentage of differences between the lengths of the ASeq generated by the exact algorithm, ExAS and non-exact algorithms, SAAS and CFAS, increases with the increase in the number of elements in the set *N*. For example, for fixed *e*=7 and *k*=2, 4, 6, 8, and 10, the percentages of cases that the exact algorithm generates ASeq with a length less than that generated by the SAAS algorithm are 12%, 40%, 56%, 76%, and 88%. Similarly, for the CFAS algorithm, the differences are 28%, 64%, 76%, 84%, and 92%.

Second, the last column in Table 1 shows the comparison between the lengths of ASeqs generated by the SAAS and CFAS algorithms. The data in Table 1 shows that the SAAS algorithm outperforms the CFAS algorithm in terms of the length of generated ASeq for all studied cases. It is important to point out that the SAAS algorithm guarantees that the generated ASeq has a length less than or equal to that generated by the CFAS algorithm.

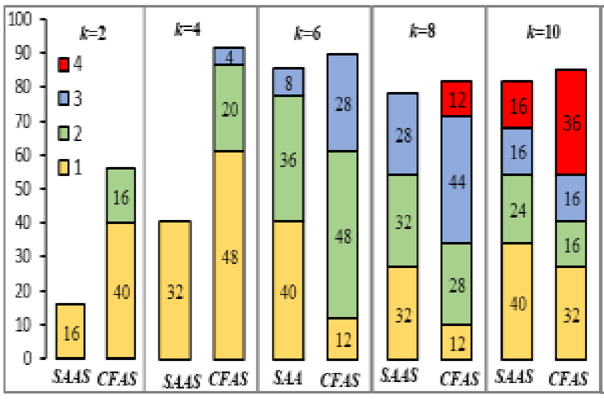
Third, the length of ASeq generated by the SAAS algorithm is near the minimal length compared to that generated by the CFAS algorithm. Figure 2 shows the distribution of the difference between the length of the shortest ASeq generated by the exact algorithm and the lengths of the generated ASeq using the SAAS and CFAS algorithms.



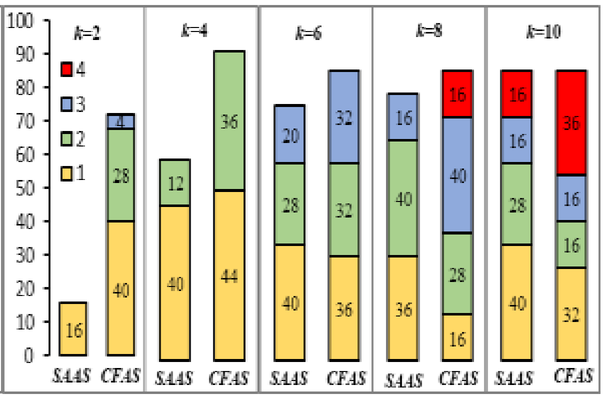
(*a*) *e*=7



(*b*) *e*=8



(*c*) *e*=9



(*d*) *e*=10

Figure 2. Percentage of differences in terms of the length of ASeq for the cases: (i) ExAS & SAAS, and (ii) ExAS & CFAS. The bar in the figure contains four colors at maximum. The gold, green, blue and red colors represent the percentage of cases that have difference equal to 1, 2, 3, and 4, respectively. The figure includes four subfigures in case of (a) e=7, (b) e=8, (c) e=9, and (d) e=10. Each subfigure contains five pairs of bars, one for SAAS algorithm and the other for CFAS algorithm. The five pairs of bars represent the five cases k=2,4, 6, 8, and 10.

Fourth, Figure 3 shows how the lengths of ASeqs change during the execution time of the SAAS algorithm for five instances in the case of *e*=10 and *k*=4. The SAAS algorithm starts with a short ASeq and then finds the shortest ASeq.

It is clear that the SAAS algorithm generates short *ASeqs* with lengths that are closer to the shortest *ASeq* than those generated by the CFAS algorithm. For example, when *e=8* and *k=2*, there are 20% of the instances where the length of *ASeq* generated by the SAAS algorithm is longer by one than the length of *ASeq* generated by the ExAS algorithm. On the other side, using the CFAS algorithm, 44% and 20% of instances have lengths greater than the shortest by one and two, respectively.

Fifth, the proposed algorithm has run on the same instances but with a change in the value of γ between 0.8 and 0.99, and the results have not changed, i.e., the generated ASeq has the same length. Similarly, when we change the initial value of *T* to 20, 25 and 30.

The time comparison between the three algorithms, ExAS, SAAS, and CFAS is shown in Table 2. The analysis of the data results demonstrates the following notes. (1) The fastest running time for all compared algorithms is CFAS algorithm. (2) In general, the values of *e* and *k* have no impact on the CFAS algorithm. On the other side, the SAAS algorithm is slightly affected by increasing *e* and *k*, whereas the ExAS algorithm is significantly affected by increasing *e* and *k*. (3) The execution time for the SAAS algorithm is affected by the two parameters, *succNo* and *metropolisNo*. The value of the running time for SAAS algorithm increases slightly with increasing the values of two parameters. (4) The execution time for the SAAS algorithm is faster than the exact algorithm, and the difference between the two algorithms in running time increases with an increase in *e* and *k*. (5) The last column of Table 2 illustrates the percentage improvement for the SAAS algorithm against the ExAS algorithm.

Table 2. Comparison between different algorithms in terms of running time in milliseconds.

e	k	ExAS Alg.	SAAS Alg.	CFAS Alg.	% of improvement SAAS & ExAS
	2	10	65	1	---
	4	12	76	1	---
7	6	14	89	1	---
	8	16	95	2	---
	10	17	99	2	---
	2	12	82	1	---
	4	107	101	2	5.5%
8	6	175	112	2	35.9%
	8	245	115	3	53.3%
	10	307	116	4	62.1%
	2	13	107	2	---
	4	423	131	2	68.9%
9	6	4376	145	3	96.7%
	8	14782	158	4	98.9%
	10	46592	162	4	99.7%
	2	15	147	4	---
	4	57827	167	4	99.7%
10	6	805166	178	16	99.9%
	8	15878846	186	16	100%
	10	58645310	197	18	100%

Note that: it should be pointed out that the implementation of the ExAS algorithm [19] used the CFAS algorithm to generate an upper bound of $\ell(N)$, i.e., if the ExAS algorithm does not find a shortest ASeq with a length less than the length generated by the CFAS algorithm, then we stop the search and take the generated ASeq by the CFAS algorithm as a shortest ASeq. This technique improves the running times of the ExAS algorithm dramatically. This explains why the running time for the ExAS algorithm is small for small data sets, as shown in Table 2 in the case of *e*=7.

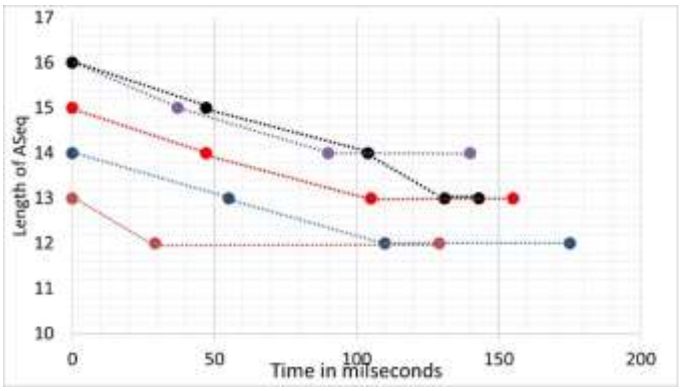


Figure 3. Change of the length of ASeq over execution time for five instances.

5. Conclusions and future works

We have proposed a new metaheuristic algorithm to find an addition sequence with a short length for a set of positive numbers *N*. The proposed algorithm starts with generating an addition sequence for *N* using the CFAS algorithm and then applies the simulated annealing strategy to get an addition sequence for *N* with shorter length. The proposed algorithm is very fast compared to the exact algorithm and can generate an addition sequence with a shorter length than the previous suboptimal algorithms.

The efficiency of the proposed algorithm is determined by considering different parameters, such as the number of elements in the set *N* and the domain of the elements of the set *N*.

There are many research directions related to this study that can be done in the future, such as: (1) How to apply the same (or similar) technique to B-chains and vectorial addition chains. (2) How to accelerate the computation of ASeq using high-performance systems. (3) How to accelerate multi-modular exponentiation using ASeq. (4) Use of some recent strategies, such as the discrete Jaya algorithm and the evolutionary programming, to solve the ASeq problem [26,27].

Use of AI tools declaration

The authors declare that they have not used Artificial Intelligence (AI) tools in the creation of this article.

Acknowledgements

The authors are grateful to the referees for their valuable comments that helped to improve the paper. The authors would like to acknowledge the support provided by Deputy for Research & Innovation, Ministry of Education through Initiative of Institutional Funding at University of Ha'il – Saudi Arabia through project number IFP-22 025.

Conflict of interest

The authors declare that they have no conflicts of interest.

References

1. D. E. Knuth, The art of computer programming: Seminumerical algorithms, **2**, 3Ed. Addison-Wesley, Reading, 461–485, (1997).
2. A. Menezes, P. van Oorschot, S, Vanstone. Handbook of applied cryptography, CRC Press, Boca Raton, 1996 (Chapter 14).
3. A. Yao, On the evaluation of powers, *SIAM J. Comput.*, **5** (1976) 100–103. https://doi.org/10.1137/0205008
4. D. Bleichenbacher, Efficiency and security of cryptosystems based on number theory, chapter 4. A Docotor Thesis, Swiss Federal Institue of Technology Zurich, Zurich, 1996. https://www.research-collection.ethz.ch/handle/20.500.11850/142613
5. R. Rivest, A. Shamir, L. Adleman, A method for obtaining digital signatures and public-key cryptosystems, *Commun. ACM,* **21** (1978), 120–126. https://doi.org/10.1145/359340.359342
6. H. Bahig, D. I. Nassr, Generating a shortest B-chain using multi-gpus, *Inf. Sci. Lett.,* **11** (2022), 745–750. https://doi.org/10.18576/isl/110307
7. E. J. Guzmán-Trampe, N. Cruz-Cortés, L. J. Dominguez Perez, D. Ortiz-Arroyo, Francisco Rodríguez-Henríquez, Low-cost addition–subtraction sequences for the final exponentiation in pairings, *Finite Fields Th. App.,* **29** (2014), 1–17, https://doi.org/10.1016/j.ffa.2014.02.009
8. E. Thurber, N. Clift, Addition chains, vector chains, and efficient computation, *Discret. Math.*, **344** (2021), 112200. https://doi.org/10.18576/isl/110307
9. P. Downey, B. Leong, B. R. Sethi, Computing sequences with addition chains, *SIAM J. Comput.,* **10** (1981), 638–646. https://doi.org/10.18576/isl/110307
10. C. Laih, S. Yen, L. Harn, Two efficient server-aided secret computation protocols based on the addition sequence, In: *Advances in cryptology-ASIACRYPT'91*, 450–459, 1991. https://doi.org/10.18576/isl/110307
11. C. Laih, S. Yen, Secure addition sequence and its applications on the server-aided secret computation protocols, In: *Advances in cryptology-AUSCRYPT'92*, Lecture Notes in Computer Science, **718** (1992), 219–229. https://doi.org/10.1007/3-540-57220-1_64
12. P. Nguyen, I. E. Shparlinski, On the insecurity of a server-aided RSA protocol. In: Boyd, C. (eds) Advances in Cryptology—ASIACRYPT 2001. ASIACRYPT 2001. Lecture Notes in Computer Science, vol 2248, Springer, Berlin, Heidelberg. (2001). https://doi.org/10.1007/3-540-45682-1_2
13. X. Chen, J. Li, J. Ma, Q. Tang, W. Lou, New algorithms for secure outsourcing of modular exponentiations, *IEEE T. Parallel Distr.*, **25** (2014), 2386–2396, https://doi.org/10.1109/TPDS.2013.180
14. C. Bouillaguet, F. Martinez, D. Vergnaud, Cryptanalysis of modular exponentiation outsourcing protocols, *Comput. J.*, **65** (2022), 2299–2314. https://doi.org/10.1093/comjnl/bxab066
15. C. Chevalier, F. Laguillaumie, D. Vergnaud, Privately outsourcing exponentiation to a single server: Cryptanalysis and optimal constructions, Computer Security–ESORICS 2016, 261–278. https://doi.org/10.1007/978-3-319-45744-4_13
16. G. Di Crescenzo, M. Khodjaeva, D. Kahrobaei, V. Shpilrain, Delegating a product of group exponentiations with application to signature schemes, *J. Math. Cryptol.*, **14** (2020), 438–459. https://doi.org/10.1515/jmc-2019-0036
17. S. Kirkpatrick, C. D. Gelatt, M. P. Vecchi, Optimization by simulated annealing, *Science*, **220** (1983), 671–680. https://doi.org/10.1515/jmc-2019-0036

18. H. Bahig, H. Bahig, A new strategy for generating shortest addition sequences, *Computing*, **91** (2011), 285–306. https://doi.org/10.1515/jmc-2019-0036
19. F. Bergeron, J. Berstel, S. Brlek, Efficient computation of addition chains, *J. Theor. Nombres Bord.,* **6** (1994), 21–38. https://doi.org/10.5802/jtnb.104
20. H. Bahig, Y. Kotb, An efficient multicore algorithm for minimal length addition chains, *Computers*, **8** (2019), 23. https://doi.org/10.3390/computers8010023
21. K. Fathy, H. Bahig, M. Farag, Speeding up multi-exponentiation algorithm on a multicore system, *J. Egypt. Math. Society,* **26** (2018), 235–244. https://doi.org/10.21608/joems.2018.2540.1008
22. J. Bos, M. Coster, Addition chain heuristics. In: Brassard, G. (eds) Advances in Cryptology—CRYPTO' 89 Proceedings. CRYPTO 1989, *Lecture Notes in Computer Science,* 435. Springer, New York, NY. https://doi.org/10.1007/0-387-34805-0_37
23. A. Enge, W. Hart, F. Johansson, Short addition sequences for theta functions, *J. Integer Seq.*, **2** (2018), 1–34.
24. N. Nedjah, L. de Macedo Mourelle, Efficient pre-processing for large window-based modular exponentiation using ant colony, In: Khosla, R., Howlett, R. J., Jain, L. C. (eds) Knowledge-Based Intelligent Information and Engineering Systems. KES 2005. Lecture Notes in Computer Science, vol. 3684. Springer, Berlin, Heidelberg. https://doi.org/10.1007/11554028_89
25. M. Abbas, O. Gustafsson, Integer linear programming modeling of addition sequences with additional constraints for evaluation of power terms, 2023. https://doi.org/10.48550/arXiv.2306.15002
26. H. M. Bahig, K. A. Alutaibi, M. A. Mahdi, A. AlGhadhban, H. M. Bahig, An evolutionary algorithm for short addition chains, *Int. J. Adv. Comput. Sci. Appl.*, **11** (2020), 340–352. http://dx.doi.org/10.14569/IJACSA.2020.0111258
27. K. Gao, F. Yang, M. Zhou, Q. Pan, P. N. Suganthan, Flexible job-shop rescheduling for new job insertion by using discrete jaya algorithm, *IEEE T. Cybernetics,* **49** (2019), 1944–1955. https://doi.org/10.1109/TCYB.2018.2817240