Binary diagrams for storing ascending compositions

Mircea Merca

Received: date / Accepted: date

Abstract It is known that the integer partitions may be encoded as either ascending or descending compositions for the purposes of systematic generation. In this paper we give an efficient data structure for storing all ascending compositions of a positive integer. Using this structure, we improved the fastest known algorithm for generating integer partitions.

Keywords algorithm · ascending composition · integer partition · data structure

Mathematics Subject Classification (2000) 05A17 · 05C05 · 05C85 · 26D07

1 Introduction

Let \( n \) be a positive integer. A composition of \( n \) is a way of writing \( n \) as the sum of positive integers, i.e.,

\[
    n = \lambda_1 + \lambda_2 + \cdots + \lambda_k.
\]

If the order of integers \( \lambda_i \) does not matter, this representation is known as an integer partition. When \( \lambda_1 \leq \lambda_2 \leq \cdots \leq \lambda_k \), we have an ascending composition. If \( \lambda_1 \geq \lambda_2 \geq \cdots \geq \lambda_k \) then we have a descending composition. In order to indicate that \( \lambda = [\lambda_1, \lambda_2, \ldots, \lambda_k] \) is a partition of \( n \), we use the notation \( \lambda \vdash n \) introduced by G. E. Andrews [1]. For \( n \), the function giving the number of partitions is denoted by \( p(n) \).

Using strict binary tree structures, we produced in [5] the fastest known algorithm for the generation of the partitions of \( n \). The data structure for storing ascending compositions of \( n \) was called the partition strict binary tree and was created according to the following rule: the root of the partition strict binary tree

The final publication is available at comjnl.oxfordjournals.org:

http://comjnl.oxfordjournals.org/content/early/2012/08/23/comjnl.bxs111.abstract.

Department of Mathematics, University of Craiova
A. I. Cuza 13, 200585 Craiova, Romania
E-mail: mircea.merca@profinfo.edu.ro
of \( n \) is labeled with \((1, n - 1)\), the node \((x_l, y_l)\) is the left child of the node \((x, y)\) if and only if
\[
x_l = \begin{cases} x, & \text{if } 2x \leq y, \\ y, & \text{otherwise} \end{cases} \quad \text{and} \quad y_l = y - x_l, \tag{1}
\]
and the node \((x_r, y_r)\) is the right child of the node \((x, y)\) if and only if
\[
x_r = \begin{cases} x + 1, & \text{if } 2 + x \leq y, \\ x + y, & \text{otherwise} \end{cases} \quad \text{and} \quad y_r = x + y - x_r. \tag{2}
\]

To generate all the ascending compositions of \( n \) we can traverse in depth-first order the partition strict binary tree. When we reach a leaf node, we list from the path that connects the root node with the leaf node only the leaf node and the nodes that are followed by the left child.

For example, in figure 1 we can see that \((1, 5)(1, 4)(1, 3)(2, 2)(2, 0)\) is a path that connects the root node \((1, 5)\) to the leaf node \((2, 0)\). From this path, node \((1, 3)\) is deleted when listing because it is followed by the node \((2, 2)\) which is its right child. Keeping from every remained pair only the first value, we get the ascending composition \([1, 1, 2, 2]\).

Fig. 1 A partition strict binary tree of 6

Note that the partition strict binary trees were derived in [5] from tree structures presented by Lin [4] and are different from the binary tree structures introduced by Fenner and Loizou [3].

Clearly, the number of leaf nodes in the partition strict binary tree of \( n \) is equal to \( p(n) \). Therefore, the total number of nodes needed to store all the ascending compositions of \( n \) in its partition strict binary tree is \( 2p(n) - 1 \). Also, the total number of edges in the partition strict binary tree of \( n \) is \( 2p(n) - 2 \).
In this article we present a data structure that needs only $\lfloor n^2/4 \rfloor + n$ nodes and $\lfloor n^2/2 \rfloor$ edges to store all the ascending compositions of $n$ ($\lfloor x \rfloor$ denotes the largest integer not greater than $x$). It is clear that there can be a tree structure. Using this structure we present a new version of the fastest known algorithm for generating the partitions of $n$ [5, Algorithm 6].

2 Diagram structure

In figure 1, we see that the left subtree of the node $(2,4)$ is identical to the right subtree of the node $(1,3)$. This redundancy of information in the partition strict binary tree is confirmed by the following result:

**Theorem 1** Let $(x,y)$ be an inner node in the partition strict binary tree, such that $x > 1$. If $2x \leq y$, then the left subtree of $(x,y)$ is identical to the right subtree of the node $(x-1, y-x+1)$. If $2x > y$, then the left subtree of $(x,y)$ is identical to the right subtree of the node $(\lfloor y/2 \rfloor, y - \lfloor y/2 \rfloor)$.

**Proof** The proof of the theorem follows easily from the rule to create the partition strict binary tree, namely the relations (1) and (2).

According to Theorem 1, in any partition strict binary tree, the left subtree of any inner node $(x,y)$ with $x > 1$ is identical to the right subtree of another node $(x',y')$. For any inner node $(x,y)$ with $x > 1$ we delete its left subtree and then create the edge from the node $(x,y)$ to the right child of the node $(x',y')$. Thus, we transform the partition strict binary tree in a directed acyclic graph. We call this new data structure *partition binary diagram*. For instance, applying these conversions to the partition strict binary tree from figure 1, we obtain the partition binary diagram in figure 2.

*Fig. 2* A partition binary diagram of 6
It is clear that the partition binary diagram is a concise representation of the partition strict binary tree. For this reason, the node \((1, n - 1)\) will be called the root node of the partition binary diagram of \(n\). Also, the node \((k, 0), k = 1, \ldots, n,\) will be called the leaf node of the partition binary diagram of \(n\). The root node is the only node in the partition binary diagram which has the in degree zero. Any leaf nodes in the partition binary diagram has the out degree zero and any other node has the out degree equal to 2. In order to generate the ascending compositions, we will generate all paths from the root to each leaf node and then we proceed as in the partition strict binary tree.

**Theorem 2** The number of nodes in the partition binary diagram of positive integer \(n\) is \(\lfloor n^2/4 \rfloor + n\).

**Proof** The partition binary diagram is obtained by deleting the redundant subtrees of the partition strict binary tree. After doing this, we obtain a partial tree that has \(\lfloor n/2 \rfloor\) nodes in its right subtree. Thus, we deduce that the number of nodes of this partial tree is

\[
\sum_{k=1}^{n} \left(1 + \left\lfloor \frac{k}{2} \right\rfloor \right) = n + \sum_{k=1}^{n} \left\lfloor \frac{k}{2} \right\rfloor = n + \left\lfloor \frac{n^2}{4} \right\rfloor.
\]

The theorem is proved.

**Theorem 3** The number of edges in the partition binary diagram of positive integer \(n\) is \(\lfloor n^2/2 \rfloor\).

**Proof** The partial binary tree obtained after deleting the redundant subtree in the partition strict binary tree has \(\lfloor n/2 \rfloor\) nodes and only one leaf node in its right subtree. The number of edges in the right subtree of the partial binary tree is equal to the number of edges to be added in this right subtree in order to obtain the partition binary diagram. This number is \(\lfloor n/2 \rfloor - 1\). We denote by \(e_n\) the number of edges in the partition binary diagram and then we deduce that

\[
e_n = e_{n-1} + 2 \left\lfloor \frac{n}{2} \right\rfloor.
\]

Thus, we get

\[
e_n = 2 \sum_{k=1}^{n} \left\lfloor \frac{n}{2} \right\rfloor = 2 \left\lfloor \frac{n^2}{4} \right\rfloor = \left\lfloor \frac{n^2}{2} \right\rfloor.
\]

The theorem is proved.

### 3 Partition binary diagrams represented as arrays

As binary trees, the partition binary diagrams can be represented as arrays. In most situations, representing a tree with an array is not very efficient. Unfilled nodes and deleted nodes leave holes in the array, wasting memory. When deletion of a node involves moving subtrees, every node in the subtree must be moved to its new location in the array, which is time-consuming in large trees.

However, if deletions are not allowed, the array representation may be useful, especially if obtaining memory for each node dynamically is, for some reason, too
time-consuming. The partition binary diagrams are data structures that do not need to delete nodes. In the array approach, the nodes of the partition binary diagrams are stored in an array and are not linked by references. The position of the node in the array corresponds to its position in the partial tree which is obtained by deleting the redundant subtrees in the partition strict binary tree. The index of nodes is determined by the postorder traversal of the partial tree. For instance, we can see this on the partition binary diagram in figure 3.

Fig. 3 Indexing nodes in the partition binary diagram of 6

3.1 A classical approach

To represent a partition binary diagram as an array, a node is declared as a structure with an information field $\lambda$ and two pointer fields $L$ and $R$. These pointer fields contain the indexes of the array cells in which the left and right children are stored, if there are any. If the index of the node $(x,y)$ is $k$ then

$$L(k) = \begin{cases} 
\text{the index of the left child of } (x,y), & \text{for } y > 0 , \\
0, & \text{for } y = 0 ,
\end{cases}$$

$$R(k) = \begin{cases} 
\text{the index of the right child of } (x,y), & \text{for } y > 0 , \\
0, & \text{for } y = 0 ,
\end{cases}$$

$$\lambda(k) = x .$$

For instance, the partition binary diagram from figure 3 can be represented as the array in table 4. The root is always located in the last cell, cell 15, and 0 indicates
a null child. In this representation, the two children of node 14 are located in positions 7 and 13, and the left and right children of node 12 are null.

Table 1  Partition binary diagram of 6 represented as array

<table>
<thead>
<tr>
<th>Index</th>
<th>k</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
<th>12</th>
<th>13</th>
<th>14</th>
<th>15</th>
</tr>
</thead>
<tbody>
<tr>
<td>Left</td>
<td>L(k)</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>3</td>
<td>0</td>
<td>2</td>
<td>5</td>
<td>0</td>
<td>4</td>
<td>8</td>
<td>0</td>
<td>4</td>
<td>7</td>
<td>11</td>
</tr>
<tr>
<td>Right</td>
<td>R(k)</td>
<td>0</td>
<td>0</td>
<td>2</td>
<td>0</td>
<td>4</td>
<td>0</td>
<td>6</td>
<td>7</td>
<td>0</td>
<td>9</td>
<td>10</td>
<td>0</td>
<td>12</td>
<td>13</td>
<td>14</td>
</tr>
<tr>
<td>Info</td>
<td>λ(k)</td>
<td>1</td>
<td>2</td>
<td>1</td>
<td>3</td>
<td>1</td>
<td>4</td>
<td>2</td>
<td>1</td>
<td>5</td>
<td>2</td>
<td>1</td>
<td>6</td>
<td>3</td>
<td>2</td>
<td>1</td>
</tr>
</tbody>
</table>

Let \((a_n)_{n \geq 0}\) be an integer sequences defined by \(a_n = \lfloor n^2/4 \rfloor\) (the sequence A002620 in [6]). We have already shown that the number of nodes in the partition binary diagram of \(n\) is \(a_n + 2 - 1\) and the number of edges is \(2a_n\). Note that the array used to represent the partition binary diagram of \(n\) stores the ascending compositions for all \(k \leq n\). To generate the ascending compositions of \(k\), we will generate all paths from the node indexed with \(a_k + 2 - 1\) to each leaf node with the index \(a_i + 1\), \(i = 1, \ldots, k\), for all \(k \leq n\). The direct access to any node is possible in the array representation.

**Theorem 4** Let \(k\) and \(n\) be two non-negative integers such that \(0 \leq k < \lfloor n^2/4 \rfloor\). Then

\[
L(a_n - k) = \begin{cases}
0, & \text{if } k = 0, \\
a_n - k - 1, & \text{otherwise},
\end{cases}
\]

\[
R(a_n - k) = \begin{cases}
0, & \text{if } k = 0, \\
a_n - k - 1, & \text{otherwise},
\end{cases}
\]

\[
λ(a_n - k) = \begin{cases}
-1, & \text{if } k = 0, \\
k, & \text{otherwise},
\end{cases}
\]

where \(a_n = \lfloor n^2/4 \rfloor\) and \([x]\) denotes the nearest integer to \(x\), i.e., \([x] = \lfloor x + \frac{1}{2} \rfloor\).

**Proof** By Theorem 2, we deduce that \(a_{k+2} - 1\) is the index of the node \((1, k - 1)\), for all \(k \leq n\). Thus, we obtain that

\(a_{k+1}\) is the index of the leaf node \((k, 0)\)

(6)

and

\(a_{n+2} - 1 - k\) is the index of the node \((1 + k, n - 1 - k)\).

(7)

If \(2(1+k) \leq n-(1+k)\), then by Theorem 1 we deduce that the node \((1+k, n-2-2k)\) is the left child of the node \((1+k, n-1-k)\). By (7), we get that \(a_{n+1-k} - 1 - k\) is the index of \((1+k, n-2-2k)\). Also, if \(2(1+k) > n-(1+k)\) then by Theorem 1 we deduce that the node \((n-1-k, 0)\) is the left child of the node \((1+k, n-1-k)\). By (6), we get that \(a_{n-k}\) is the index of \((n-1-k, 0)\). Thus, the relation (3) is proved. The relations (4) and (5) follow easily by (6) and (7).
Note that all components of the unidimensional arrays $L$, $R$ and $\lambda$ can be determined by Theorem 4. Using the relation 
\[ a_n = a_{n-1} + \left\lfloor \frac{n}{2} \right\rfloor , \]
we deduce that
\[ \{ A_i | i = 2, 3, \ldots, n \} , \]
where
\[ A_i = \left\{ a_i - k \left| k = 0, 1, \ldots, \left\lfloor \frac{i}{2} \right\rfloor - 1 \right\} , \]
is a set partition of 
\[ \{ 1, 2, \ldots, a_n - 1 \} . \]

**Lemma 1** Let $k$ and $n$ be two non-negative integers such that $0 \leq k < \left\lfloor \frac{n}{2} \right\rfloor$. If 
\[ a_n = \left\lfloor \frac{n^2}{4} \right\rfloor , \]
then 
\[ \left\lceil \sqrt{a_n - k} \right\rceil = \left\lceil \sqrt{a_n} \right\rceil , \]
where $\lceil x \rceil$ is the smallest integer not less than $x$.

**Proof** Taking into account that 
\[ a_{2n} = n^2 \quad \text{and} \quad a_{2n+1} = n^2 + n , \]
we can write 
\[ n - 1 < \sqrt{a_{2n} - k} \leq n \quad \text{and} \quad n < \sqrt{a_{2n+1} - k} < n + 1 , \] (8)
for all $0 \leq k < n$. On the other hand, we have 
\[ \left\lceil \sqrt{a_{2n}} \right\rceil = n \quad \text{and} \quad \left\lceil \sqrt{a_{2n+1}} \right\rceil = n + 1 . \] (9)
According to (8) and (9), the lemma is proved.

**Theorem 5** Let $n$ be a non-negative integer. Then
\[ L(n) = \begin{cases} 0, & \text{if } b_n = 0 , \\ \left\lceil \frac{\sqrt{4n} - b_n}{4} \right\rceil - b_n, & \text{if } 0 < b_n < \left\lceil \frac{\sqrt{4n}}{4} \right\rfloor , \\ \left\lceil \frac{\sqrt{4n} - b_n - 1}{4} \right\rceil , & \text{otherwise} , \end{cases} \] (10)
\[ R(n) = \begin{cases} 0, & \text{if } b_n = 0 , \\ n - 1, & \text{otherwise} , \end{cases} \] (11)
\[ \lambda(n) = \begin{cases} \left\lceil \frac{\sqrt{4n}}{4} \right\rceil - 1, & \text{if } b_n = 0 , \\ b_n, & \text{otherwise} , \end{cases} \] (12)
where
\[ b_n = \left\lceil \frac{\sqrt{4n}}{4} \right\rceil - n . \]
Proof Let $t$ and $k$ be two non-negative integers such that

$$a_n - k = t \quad \text{and} \quad 0 \leq k < \left\lfloor \frac{n}{2} \right\rfloor .$$

By Lemma 1, we deduce that

$$\left\lfloor \frac{n}{2} \right\rfloor = \left\lceil \sqrt{t} \right\rceil . \quad (13)$$

Taking into account that

$$2\sqrt{t} \leq \left\lceil 2\sqrt{t} \right\rceil \leq 2\left\lceil \sqrt{t} \right\rceil ,$$

we get

$$\left\lceil \frac{2\sqrt{t}}{2} \right\rceil = \left\lceil \sqrt{t} \right\rceil . \quad (14)$$

By (13) and (14), we obtain $n = \left\lceil \sqrt{4t} \right\rceil$. Thus, the relations (10), (11) and (12) follow easily by Theorem 4.

With Theorem 5, a node’s children can be found by applying some simple arithmetic to the node’s index number in the array representation. In fact, Theorem 5 shows that the partition binary diagram can be represented with the sequences $(a_n)_{n>0}$ and $(b_n)_{n>0}$. Note that $(b_n)_{n>0}$ is the sequence A025672 in [6] and $b_{n^2} = b_{n^2+n} = 0$.

Moreover, $b_k$ is zero if and only if $k$ is a term of the sequence $(a_n)_{n>0}$.

3.2 Ascending composition matrix

Deleting redundant subtrees from the partition strict binary tree, we obtained a partial tree in which every node of the form $(x,y)$, $x > 1$ has no left child. This property allows us to rearrange the nodes of the partition binary diagram of $n$ in a two-dimensional array with $\left\lfloor \frac{n}{2} \right\rfloor + 1$ rows and $n$ columns. Each node of the binary diagram occupies a cell of the array. The cell with indexes $(i,j)$ is occupied by the node $(x,y)$ if and only if

$$i = \begin{cases} \left\lfloor \frac{x}{2} \right\rfloor + 1, & \text{if } y = 0, \\ x, & \text{otherwise} \end{cases} \quad \text{and} \quad j = x + y . \quad (15)$$

It is clear that there will be cells that are not occupied. This is what we call *ascending composition array*. The nodes of the form $(1,y)$ are always located on the first row and the leaf node $(x,0)$ is located in the cell with the indexes $(\lfloor x/2 \rfloor + 1, x)$, for all $x \leq n$. By Theorem 2, we deduce that the number of unoccupied cells in this array is the square number $\left\lfloor \frac{n}{2} \right\rfloor^2$ (the sequences A008794 in [6]).

For example, according to (15), the nodes of the partition binary diagram from figure 3 can be rearranged in figure 4. The root node is always located in the last cell of the first row, the cell with indexes $(1,6)$, and the number of unoccupied cells is 9.
To represent a partition binary diagram as an ascending composition array, a node is declared as a structure with an information field $\lambda_{i,j}$ and two pointer fields $L_{i,j}$ and $R_{i,j}$. These pointer fields contain the indexes of the array cells in which the left and right children are stored, if there are any. Taking into account (15), it is an easy exercise to deduce that

$$L_{i,j} = \begin{cases} (i, j - i), & \text{for } 3i \leq j, \\ \left\lfloor \frac{j - i}{2} \right\rfloor + 1, j - i), & \text{for } 2i \leq j < 3i \end{cases}$$

and

$$R_{i,j} = (i + 1, j), \quad \text{for } 2i \leq j.$$ (17)

If the node $(x, y)$ from the partition binary diagram has the indexes $(i, j)$ in the ascending composition array, then

$$\lambda_{i,j} = x.$$ (18)

If the cell with the indexes $(i, j)$ are not occupied, then

$$\lambda_{i,j} = 0.$$ (19)

Thus, we consider that the matrix

$$A = [\lambda_{i,j}]_{i=1,\ldots,\lfloor n/2 \rfloor+1, j=1,\ldots, n}$$

with entries

$$\lambda_{i,j} = \begin{cases} i, & \text{if } 2i \leq j, \\ j, & \text{if } i = \lfloor j/2 \rfloor + 1, \\ 0, & \text{otherwise} \end{cases}$$

is a concise representation of the partition binary diagram of $n$ and we call it the *ascending composition matrix*.
For instance, the ascending composition matrix of 10 is
\[
\begin{bmatrix}
1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\
0 & 2 & 3 & 2 & 2 & 2 & 2 & 2 & 2 & 2 \\
0 & 0 & 0 & 4 & 5 & 3 & 3 & 3 & 3 & 3 \\
0 & 0 & 0 & 0 & 6 & 7 & 4 & 4 & 4 & 4 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 8 & 9 & 5 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 10
\end{bmatrix}
\]

4 Efficient algorithms for generating ascending compositions

To obtain all the ascending compositions of \( n \) we can generate all paths from the root node to the leaf nodes in the partition binary diagram. To do this we can convert efficient algorithms for traversing of the partition strict binary trees. In [5] we described and analyzed Algorithms 2 and 3 for inorder traversal of the partition strict binary trees. To convert these algorithms we use the ascending composition array representation of the partition binary diagram and the following result:

**Lemma 2** Let \( t \) be a positive integer. If the inner node \((x, y)\) from the partition binary diagram has the indexes \((i, j)\) in the ascending composition array then
\[
t \cdot x \leq y \quad \text{if and only if} \quad (t + 1) \cdot i \leq j.
\]

**Proof** The proof follows easily by (15).

By [5, Algorithm 2], we get the first algorithm for generating all paths from the root node to the leaf nodes in the partition binary diagram represented as ascending composition array, Algorithm 1.

**Algorithm 1** Generating all paths from root to leaves - version 1

```
Require: n
1: k ← 1
2: (i, j) ← (1, n)
3: while k > 0 do
4:   while 3k ≤ j do
5:     (a_k, b_k) ← (i, j)
6:     j ← j − i
7:     k ← k + 1
8:   end while
9:   t ← k + 1
10: while 2t ≤ j do
11:     (a_t, b_t) ← (i, j)
12:     (a_t, b_t) ← \( \left\lfloor \frac{j-i}{2} \right\rfloor + 1, j - i \)
13:     visit \((a_1, b_1), \ldots, (a_t, b_t)\)
14:     i ← i + 1
15: end while
16: (a_k, b_k) ← (i, j)
17: visit \((a_1, b_1), \ldots, (a_k, b_k)\)
18: k ← k − 1
19: (i, j) ← (a_k + 1, b_k)
20: end while
```
Thus, the first algorithm for generating ascending compositions follows immediately from Algorithm 1.

**Algorithm 2 Generating ascending compositions - version 1**

Require: \( n \)

1: \( k \leftarrow 1 \)
2: \( i \leftarrow 1 \)
3: \( j \leftarrow n \)
4: while \( k > 0 \) do
   5:      while \( 3i \leq j \) do
      6:         \( a_k \leftarrow i \)
      7:         \( j \leftarrow j - i \)
      8:         \( k \leftarrow k + 1 \)
      9:      end while
   10:     \( t \leftarrow k + 1 \)
   11:    while \( 2i \leq j \) do
      12:       \( a_k \leftarrow i \)
      13:       \( a_t \leftarrow j - i \)
      14:       visit \( a_1, a_2, \ldots, a_t \)
      15:       \( i \leftarrow i + 1 \)
   16:    end while
   17:   \( a_k \leftarrow j \)
   18:   visit \( a_1, a_2, \ldots, a_k \)
   19:   \( k \leftarrow k - 1 \)
   20:   \( i \leftarrow a_k + 1 \)
   21:   \( j \leftarrow j + i - 1 \)
   22: end while

To present the performance of Algorithm 2, we use the integer \( p(t)(n) \) introduced in [5, section 2]: the number of the partitions \( \lambda \vdash n \) that have the property

\[
\lambda_1 \geq t \cdot \lambda_2 \quad \text{and} \quad \lambda_2 \geq \lambda_3 \geq \cdots \geq \lambda_k ,
\]

where \( t \) is a positive integer such that \( t \leq n \).

**Theorem 6** Algorithm 2 executes \( 3p(n) + 5p^{(2)}(n) \) assignment statements and evaluates \( p(n) + 3p^{(2)}(n) \) boolean expressions.

**Proof** Taking into account (16), (17) and Lemma 2, the proof is similar to the proof of [5, Theorem 5].

Taking into account that \( p^{(2)}(n) \leq p(n) \), by Theorem 6 we deduce that Algorithm 2 performs fewer assignment statements than Algorithm 5 in [5, section 4]. Both algorithms evaluate the same number of logical expressions.

By [5, Algorithm 3], applying the relations (16), (17) and Lemma 2, we obtain the second algorithm for generating all paths from the root node to the leaf nodes in the partition binary diagram represented as ascending composition array, namely Algorithm 3. Then, by Algorithm 3, we get Algorithm 4 for generating the ascending compositions.
Algorithm 3 Generating all paths from root to leaves - version 2

Require: \( n \)

1: \( k \leftarrow 1 \)
2: \((i, j) \leftarrow (1, n)\)
3: while \( k > 0 \) do
4: while \( 4i \leq j \) do
5: \((a_k, b_k) \leftarrow (i, j)\)
6: \( j \leftarrow j - i \)
7: \( k \leftarrow k + 1 \)
8: end while
9: \( t \leftarrow k + 1 \)
10: \( a \leftarrow t + 1 \)
11: while \( 3i \leq j \) do
12: \((a_k, b_k) \leftarrow (i, j)\)
13: \((p, q) \leftarrow (i, j - i)\)
14: repeat
15: \((a_t, b_t) \leftarrow (p, q)\)
16: \((a_u, b_u) \leftarrow \left(\left\lfloor \frac{q - p}{2} \right\rfloor + 1, q - p\right)\)
17: visit \((a_1, b_1), \ldots, (a_u, b_u)\)
18: \( p \leftarrow p + 1 \)
19: until \( 2p > q \)
20: \((a_t, b_t) \leftarrow \left(\left\lfloor \frac{q}{2} \right\rfloor + 1, q\right)\)
21: visit \((a_1, b_1), \ldots, (a_t, b_t)\)
22: \( i \leftarrow i + 1 \)
23: end while
24: while \( 2i \leq j \) do
25: \((a_k, b_k) \leftarrow (i, j)\)
26: \((a_t, b_t) \leftarrow \left(\left\lfloor \frac{q}{2} \right\rfloor + 1, j - i\right)\)
27: visit \((a_1, b_1), \ldots, (a_t, b_t)\)
28: \( i \leftarrow i + 1 \)
29: end while
30: \((a_k, b_k) \leftarrow (i, j)\)
31: visit \((a_1, b_1), \ldots, (a_k, b_k)\)
32: \( k \leftarrow k - 1 \)
33: \((i, j) \leftarrow (a_k + 1, b_k)\)
34: end while

Theorem 7 Algorithm 4 executes \( 3p(n) + 2p^{(2)}(n) + 4p^{(3)}(n) \) assignment statements and evaluates \( p(n) + 4p^{(3)}(n) \) boolean expressions.

Proof Taking into account (16), (17) and Lemma 2, the proof is similar to the proof of [5, Theorem 6].

Theorem 8 Algorithm 4 is more efficient than Algorithm 2.

Proof Taking into account Theorems 6 and 7, the proof is similar to the proof of [5, Theorem 7].

One can see that Algorithm 4 executes fewer assignment statements than Algorithm 6 which we have presented in [5, section 4] and both algorithms evaluate the same number of boolean expressions. We denote by \( r_a(n) \) the ratio of the number of assignment statements executed by Algorithm 4 and the number of assignment statements executed by [5, Algorithm 6], i.e.,

\[
r_a = \frac{3p(n) + 2p^{(2)}(n) + 4p^{(3)}(n)}{4p(n) + 5p^{(3)}(n)}.
\]
Taking into account that
\[
\lim_{n \to \infty} \frac{p^{(2)}(n)}{p(n)} = 0 \quad \text{and} \quad \lim_{n \to \infty} \frac{p^{(3)}(n)}{p(n)} = 0,
\]
we obtain
\[
\lim_{n \to \infty} r_n(n) = \frac{3}{4}.
\]

**Algorithm 4 Generating ascending compositions - version 2**

**Require:** \( n \)
1. \( k \leftarrow 1 \)
2. \( i \leftarrow 1 \)
3. \( j \leftarrow n \)
4. \( \text{while} \ k > 0 \text{ do} \)
5. \( \quad \text{while} \ 4i \leq j \text{ do} \)
6. \( \quad a_k \leftarrow i \)
7. \( \quad j \leftarrow j - i \)
8. \( \quad k \leftarrow k + 1 \)
9. \( \quad \text{end while} \)
10. \( t \leftarrow k + 1 \)
11. \( u \leftarrow t + 1 \)
12. \( \text{while} \ 3i \leq j \text{ do} \)
13. \( \quad a_k \leftarrow i \)
14. \( \quad p \leftarrow i \)
15. \( \quad q \leftarrow j - i \)
16. \( \quad \text{repeat} \)
17. \( \quad a_t \leftarrow p \)
18. \( \quad a_u \leftarrow q - p \)
19. \( \quad \text{visit} \ a_1, a_2, \ldots, a_u \)
20. \( \quad p \leftarrow p + 1 \)
21. \( \quad \text{until} \ 2p > q \)
22. \( \quad a_t \leftarrow q \)
23. \( \quad \text{visit} \ a_1, a_2, \ldots, a_t \)
24. \( \quad i \leftarrow i + 1 \)
25. \( \quad \text{end while} \)
26. \( \text{while} \ 2i \leq j \text{ do} \)
27. \( \quad a_k \leftarrow i \)
28. \( \quad a_t \leftarrow j - i \)
29. \( \quad \text{visit} \ a_1, a_2, \ldots, a_t \)
30. \( \quad i \leftarrow i + 1 \)
31. \( \quad \text{end while} \)
32. \( a_k \leftarrow j \)
33. \( \text{visit} \ a_1, a_2, \ldots, a_k \)
34. \( k \leftarrow k - 1 \)
35. \( i \leftarrow a_k + 1 \)
36. \( j \leftarrow j + i - 1 \)
37. \( \text{end while} \)

Now let us measure CPU time for few value of \( n \). To do this we encode the algorithms in C++ and the programs obtained with Visual C++ 2010 Express Edition will be run on a computer with Intel Pentium Dual processor. CPU time is measured when the program runs without printing out ascending compositions. We denote by \( t_1(n) \) the average time for Algorithm 2 obtained after ten measurement, by \( t_2(n) \) the average time for [5, Algorithm 6] obtained after ten measurement,
and by \( r(n) \) the ratio of \( t_1(n) \) and \( t_2(n) \), i.e., \( r(n) = t_1(n)/t_2(n) \). In Table 2 we present the results obtained after the measurements made. We note that \( r(n) \leq 1 \).

### Table 2

Experimental results for \( r(n) \)

<table>
<thead>
<tr>
<th>( n )</th>
<th>( 80 )</th>
<th>( 90 )</th>
<th>( 100 )</th>
<th>( 105 )</th>
<th>( 110 )</th>
<th>( 115 )</th>
<th>( 120 )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( r(n) )</td>
<td>1.0000</td>
<td>0.9646</td>
<td>0.9594</td>
<td>0.9493</td>
<td>0.9516</td>
<td>0.9517</td>
<td>0.9515</td>
</tr>
</tbody>
</table>

5 Observations and conclusions

Theorem 7 presented in [5] required a proof of the following inequality:

\[
p(n) - p(n - 1) - p(n - 2) + p(n - 5) \leq 0 , \quad n > 0 .
\]

This inequality is presented in [2] as a special case of an infinite family of inequalities for the partition function \( p(n) \). Once we produce Algorithm 2, we did an experiment and we was surprised that Algorithm 2 performs fewer assignment statements than Algorithm 6 presented in [5]. So, we found a new inequality involving the partition function \( p(n) \):

**Conjecture 1** Let \( n \) be an integer. The inequality

\[
p(n) - 5p(n - 3) + 5p(n - 5) \geq 0
\]

is true if and only if \( n \neq 3 \).

If we need a simple and efficient algorithm for generating ascending compositions in lexicographic order then Algorithm 2 or [5, Algorithm 5] can be the best choice. If we need the fastest algorithm to do this, then Algorithm 4 or [5, Algorithm 6] will be a better choice.

A representation for the integer partitions of \( n \) as an ascending composition array has been introduced. The space and time complexity for creating the ascending composition array of \( n \) is only \( O(n^2) \). Moreover, the ascending composition array of \( n \) can be used for all positive integers less than or equal to \( n \). Thus, a much larger number of integer partitions will be stored.

**Acknowledgements** The author would like to express his gratitude for the careful reading and helpful remarks to Oana Merca, which have resulted in what is hopefully a clearer paper.

**References**