DATA STRUCTURES AND ALGORITHMS FOR PARTITIONING A SET INTO SETS
OF NON-DESCENDING CARDINALITY

A THESIS IN
Computer Science

Presented to the Faculty of the University
Of Missouri Kansas City In partial fulfillment
Of the requirements for the degree

MASTER OF SCIENCE

By
OSHANI TITTI

B.Tech. ANDHRA UNIVERSITY, INDIA 2013

Kansas City, Missouri
2015
DATA STRUCTURES AND ALGORITHMS FOR PARTITIONING A SET INTO SETS
OF NON-DESCENDING CARDINALITY

Oshani Titti, Candidate for the Master of Science Degree
University of Missouri – Kansas City, 2015

ABSTRACT

Data structures have been around since the structured programming era. Algorithms often associate with data structures. An algorithm is a sequence of instructions that accomplishes a task in a finite time period. The algorithm receives zero or more inputs, produces at least one output, consists of clear and unambiguous instructions, terminates after a finite number of steps, and is basic enough that a person can carry out the algorithm using a pencil and paper. Algorithms for dividing objects into bins have long been invented. However, dividing objects in summation format is not received due attention. In this paper, objects are divided into n bins in such a way that the next bin will contain more than or equal number of objects than the preceding bin.

Keywords: Data structure, algorithms, edge partition, integer partition, non-descending order partition, edge cover
APPROVAL

The faculty listed below, appointed by the Dean of School of Computing and Engineering, have examined a thesis titled “Data Structures and Algorithms for Partitioning a Set into Sets of Non-Descending Cardinality” presented by Oshani Titti, candidate for the Master of Science degree, and certify that in their opinion it is worthy of acceptance.

Supervisory Committee

Yijie Han, Ph.D., Chair
School of Computing and Engineering

Vijay Kumar, Ph.D.
School of Computing and Engineering

Appie van de Liefvoort, Ph.D.
School of Computing and Engineering
CONTENTS

ABSTRACT ................................................................. iii
ILLUSTRATIONS ............................................................. vii
TABLES ................................................................. viii
ACKNOWLEDGEMENTS. ................................................... ix
INTRODUCTION. ......................................................... 1

Chapter

1. GENERATING EDGE COVERS OF PATH ......................... 3
   1.1 Introduction ....................................................... 3
   1.2 Partitions of Integers ............................................ 4
   1.3 Example .......................................................... 4
   1.4 Algorithm for Generating Ascending Composition of an Integer n in m parts. . . . 5
   1.5 Binary Strings to Represent Edge Covers ..................... 7

2. EFFICIENT DATA STRUCTURES FOR STORING THE PARTITIONS OF
   INTEGERS ............................................................ 11
   2.1 Introduction ....................................................... 11
   2.2 Four Data Structures ............................................ 11
   2.3 Linear Structures ................................................ 12
   2.4 Tree Structure ................................................... 13
   2.5 Diagram Structure .............................................. 16

3. APPROACH ............................................................. 19
ILLUSTRATIONS

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.1</td>
<td>Partition Diagram for Integer 6.</td>
<td>5</td>
</tr>
<tr>
<td>1.2</td>
<td>An Algorithm for Generating Ascending Composition n in m Parts</td>
<td>6</td>
</tr>
<tr>
<td>2.1</td>
<td>Tree Structure for Integer 6.</td>
<td>13</td>
</tr>
<tr>
<td>2.2</td>
<td>A Partition Tree for Integer n.</td>
<td>14</td>
</tr>
<tr>
<td>2.3</td>
<td>An Algorithm for Creating a Partition Tree.</td>
<td>15</td>
</tr>
<tr>
<td>2.4</td>
<td>A Partition Tree with Data Structure Sharing.</td>
<td>17</td>
</tr>
<tr>
<td>3.1</td>
<td>Partition Diagram to Divide 17 Objects into 6 bins</td>
<td>23</td>
</tr>
<tr>
<td>3.2</td>
<td>Algorithm for Non-Descending Partition</td>
<td>25</td>
</tr>
<tr>
<td>3.3</td>
<td>Different ways of Arranging the Objects</td>
<td>25</td>
</tr>
<tr>
<td>3.4</td>
<td>Different ways of Adding an Object in the above Arrangement</td>
<td>26</td>
</tr>
<tr>
<td>3.5</td>
<td>Arrangement of Objects to Achieve Maximum Steps.</td>
<td>27</td>
</tr>
</tbody>
</table>
## TABLES

<table>
<thead>
<tr>
<th>Table</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. Kernel Strings</td>
<td>9</td>
</tr>
</tbody>
</table>
ACKNOWLEDGEMENTS

I would like to take this opportunity to thank the following people who have directly or indirectly helped me in academic achievements. Firstly, I would like to thank Dr. Yijie Han, my mentor and advisor, for his continuous support and guidance throughout my master’s program in computer science. Without his tireless effort and advice, this personal achievement wouldn’t be possible. I sincerely thank Dr. Vijay Kumar and Dr. Appie van de Liefvoort for accepting to be a part of my thesis committee and making time for me from their busy schedule.

I would like to dedicate my thesis to my parents who constantly inspired me to pursue higher studies. I would like to thank my family, my dad Mr. Vivekananda Titti, my mom Mrs. Bhavani Titti and my sister Miss. Santhoshi Tejaswini, who stood behind me all these years during my degree.

Finally I would like to thank all my teachers, educational administrators, present and past and all who helped me achieve this academic goal.
INTRODUCTION

Computer science emphasizes two important topics: data structures and algorithms. Those topics are important because the choices you make for a program's data structures and algorithms affect that program's memory usage (for data structures) and CPU time (for algorithms that interact with those data structures).

This thesis initiates a two-part series that explores data structures and algorithms. When choosing a data structure or algorithm, you sometimes discover an inverse relationship between memory usage and CPU time: the less memory a data structure uses, the more CPU time associated algorithms need to process the data structure’s data items, which are primitive type values or objects, via references. Also, the more memory a data structure uses, the less CPU time associated algorithms need to process the data items—and faster algorithms result. This paper begins with a presentation of basic concepts and continues with a tour of the array data structure.

I studied two research papers. In the paper “Generating edge covers of path graphs” by J. Raymundo Marcial-Romero, J. A. Hernández, Vianney Muñoz-Jiménez and Héctor A. Montes-Venegas, I reviewed path graphs, edge covers and the concept of partitions. Path graph or linear graph is a simple example of a tree, namely a tree with two or more vertices that is not branched at all, that is, contains only vertices of degree 2 and 1. An edge cover of an undirected graph is a subset of its edges such that every vertex of the graph is incident to at least one edge of the subset. An edge cover of an undirected graph is a subset of its edges. It portrays that the set of edge covers of a given path graph can be generated by what we call a set of kernel.
In the paper “Efficient Data Structures for Storing the Partitions of Integers” by Rung-Bin Lin, Yuan Ze University, Chung-Li, 320 Taiwan, I studied four data structures for storing all the partitions of an integer. The most efficient data structure need only store integers if n is even or if n is odd. It is created without exhaustively enumerating all the partitions. The time complexity for creating this data structure is the same as the space complexity. The complexity is low enough to make possible for storing all the partitions of an integer up to several ten thousands. We have used the concept of partitions and edge covers to develop our approach of partitioning objects into sets of non-descending cardinality. The data structure discussed in the reviewed paper can be used for storing all the partitions. In our approach we have studied about partitioning a set into non-descending cardinality. The space and time complexity for creating a linear structure or a partition tree is proportional to the number of partitions. This method allows us to create a partition diagram that can store all the partitions of an integer up to several ten thousands.

I did research on finding alternative schemes for storing and generating kernel strings. I contributed an algorithm for generating kernel strings. This algorithm is more intuitive and constructive. Using this algorithm I then constructed a generation tree which has paths to leaves that represent kernel strings. Because there is no closed formula for partitioning objects non-descendingly into bins I also studied upper bounds for the number of non-descending partitions. By my analysis I reached formula \((\sqrt[n]{m})^{\sqrt{m}}\) for partitioning m objects non-descendingly to n bins, which is a trade off compared to the straight forward estimation of \((m/n)^n\).
CHAPTER 1
GENERATING EDGE COVERS OF PATH

In this chapter, we review the paper “Generating edge covers of path graphs” by J. Raymundo Marcial-Romero, J. A. Hernández, Vianney Muñoz-Jiménez and Héctor A. Montes-Venegas.

1.1 Introduction

This paper deals edge covers. An edge cover of an undirected graph is a subset of its edges such that every vertex of the graph is incident to at least one edge of the subset. Even for path graphs with m edges it has been shown that the set of edge covers is equal to fibonacci (m). As a consequence, generating the set of edge covers of a given path graph is an exponential combinatorial problem. This paper portrays that the set of edge covers of a given path graph can be generated by what we call a set of kernel. It is well known that the edge cover problem is #P complete [1]. So, heuristic and exact methods have been proposed to count the number of edge covers for classes of graphs.

Although there is a polynomial time algorithm which computes the number of edge covers of a given path graph, generating the set of edge covers of a given path graph is an exponential combinatorial problem.

The set of edge covers for path graphs is generated using an efficient algorithm for generating ascending compositions of an integer n in m parts based in the diagram structure proposed by [1,3]. The technique uses a binary pattern to map integer partitions into binary strings to represent edge covers.
1.2 Partitions of Integers

Let \( n \) be a positive integer. A composition of \( n \) is a way of writing \( n \) as the sum of positive integers denoted as \( n = y_1 + y_2 + \ldots + y_k \). If the order of integers \( y_i \) is irrelevant, this representation is an integer partition. When \( y_1 \leq y_2 \leq \ldots \leq y_k \) we have an ascending composition.

The partition diagram of an integer \( n \) is a directed acyclic graph.

1.3 Example

The partition diagram shown in Figure 1.1 is created for integer \( i = 6 \), it also consists of all the partition diagrams of any integer smaller than 6. The partition diagram of any integer \( r \) smaller than \( n \) is anchored at node \((1, r)\). For example, the node \((1, 5)\) is the anchored node of the partition diagram of 5.

The partition is formed during a path traversal by recovering all the first parts of the tuples excluding the anchored node and nodes of the form \((1, r)\) if its predecessor is of the form \((y, r)\), where \( y > 1 \). For example, the path \((1, 6) (1, 5) (1, 4) (2, 2) (1, 2) (2, 0)\) defines partition \((1, 1, 2, 2)\) since the nodes \((1, 6)\) and \((1, 2)\) are excluded. Traversing all the paths, the partitions of 6 are \((1, 1, 1, 1, 1, 1)\), \((1, 1, 1, 1, 2)\), \((1, 1, 1, 3)\), \((1, 1, 2, 2)\), \((1, 1, 4)\), \((1, 2, 3)\), \((1, 5)\), \((2, 2, 2)\), \((2, 4)\), \((3, 3)\), and \((6)\).
1.4 Algorithm for generating ascending composition of an integer n in m parts

To obtain all the ascending composition of integer n in m parts, we can generate all paths from the root node to the leaf nodes whose depth is m + 1 using the partition diagram. To achieve this, they present a variant of Merca algorithm for traversing all the paths with the constraint of m parts. We represent each level of the partition diagram as a dynamic vector, i.e. diagram[0] = {(1, 6)}, diagram[1] = {(1, 5), (2, 4), (3, 3), (6, 0)} and so on. Particular elements of the diagram can be recovered as diagram[row][column] where row represent the level of the diagram and column the position in the level.

The required variables should be initialized as follows. Variables row ← 1 and column ← 0, these values represent their position in the diagram. The variable part ← 0, is the number of parts in which n has been already divided. Finally, partition ← {} has the elements of a
partition. When a partition is formed, the set partition is stored and modified to generate a new partition.

Algorithm: Parts (Generating ascending composition n in m parts)
Require: diagram (Diagram structure of n)
Require: m, row, column, part, partition

for all i = 0; i ≤ length(diagram [row] −1); i = i+ 1 do
    partition = partition ∪ first component (diagram [row] [i]);
    part ← part + 1;
    if second component (diagram [row] [i])! = 0 and part < m then
        Parts (n − second component (diagram [row] [i]) + 1, 0, part, m, n)
    else
        if second component (diagram [row] [i] = 0) and part == m then
            store partition;
        end if
    end if
end for

Figure 1.2: An Algorithm for Generating Ascending Composition n in m Parts.
Binary Strings to Represent Edge Covers

A binary pattern can be used to represent an edge cover of a path graph. Let \( b_1 b_2 \ldots b_m \) be a binary sequence. If \( b_i = 1 \) then the edge \((a_i, a_{i+1})\) belongs to the edge cover otherwise (i.e. \( b_i = 0 \)) the edge \((a_i, a_{i+1})\) does not belong to the edge cover. Let \( G \) be a path graph with \( n \) nodes. A binary sequence \( b_1 b_2 \ldots b_m, m = n-1 \) represents an edge cover for \( G \) if the following conditions hold:

1) \( \not\exists b_i, b_{i+1} \) such that \( b_i = 0 \) and \( b_{i+1} = 0 \).
2) \( b_1 = 1 \) and \( b_m = 1 \).

An edge cover representation, does not admit a sequence of consecutive zeros. So, a binary sequence is represented as

\[ 1^{p_0}01^{p_1} \cdot \cdot \cdot 01^{p_{l-1}}1^{p_l} \]

for some \( l \in \mathbb{N} \) and \( p_0, p_1, \ldots p_l > 0 \).

Lemma 1:

Let \( \omega \) be a binary sequence which representation edge cover of a path graph with \( n \) nodes. The number \( s_n \) of zeros appearing in \( \omega \) is given by [1]

\[
S_n = \begin{cases} 
(n-2)/2; & \text{if } n \text{ is even} \\
(n-1)/2; & \text{if } n \text{ is odd.}
\end{cases}
\]

Proof:

It is obvious that, when each \( p_i = 1, 1 \leq i \leq n \), the string \( \omega \) has the maximum number of zeros. Since the one’s appearing at the extrema of \( \omega \) are fixed, then a string of length \( n - 2 \) is left where half of the symbols are zero if \( n \) is even or half of \( n - 1 \) symbols are zero if \( n \) is odd. ■

Corollary 1: Let \( P \) be a path graph such that the number of nodes of \( P \) is \( n \). If \( \omega \) is a binary string which represents an edge cover for \( P \) then
1) If $n$ is even there are $0 \leq s \leq (n-2)/2$ zeros on $\omega$;

2) If $n$ is odd there are $0 \leq s \leq (n-1)/2$ zeros on $\omega$;

Lemma 2: Let $P$ be a path graph with $m$ edges ordered as $a_1, a_2, \ldots, a_m$. There are $m-2$ strings of length $m$ with one zero that represent edge covers of $P$. [1]

Proof: Let $b_1b_2\ldots b_m$ be an arbitrary string which represent the edges of the path graph $P$. It is obvious that $b_1$ and $b_m$ should be one since each of them cover the nodes $a_1$ and $a_n$ respectively. So the string where $b_2 = 0$ and $b_i = 1$, $i = 1, 3, 4, \ldots, m$ represents an edge cover of $P$ with one zero. If the value of $b_2$ is shifted to $b_3$ and assign to $b_2$ the value one, then the second string represents an edge cover of $P$ with one zero. In general if $b_i = 0$, $2 \leq i \leq m-2$ then shifting the value of $b_i$ with $b_{i+1}$ gives a new edge cover of $P$ with one zero.

The generation of some strings with more than one zero which represent edge covers can be determined via the correspondence with the integer partitions of a number $n$.

Definition 1: A kernel string is a binary string of the form

$01^{p_1}01^{p_2}\ldots1^{p_l}0$, $l \geq 1$ where $0 < p_1 \leq p_2 \leq \cdots \leq p_l$.

Proposition 1: There are $n-4$ kernel strings with two zeros of length at most $n-2$.

Proof: A kernel string with two zeros is of the form $01^{p_1}0$. In fact, if $1 \leq p_1 \leq n-4$ the kernel strings $01^{p_1}0$ have length at most $n-2$.

Lemma 3:

Let $p(n,m)$ be the set of ascending integer partitions of $n$ in $m$ parts. If $l_1 + l_2 + \cdots + l_m \in p(n, m)$, then $01^{l_1}01^{l_2}\cdots1^{l_m}0$ is a kernel string with $m+1$ zeros whose length is $l_1 + l_2 + \cdots + l_m + m + 1$. 

8
Proof: That 01^{11} 01^{12} \cdots 1^{1^M} 0 is a kernel string is easily verified since each $l_i = 0$ and the ascending condition means that $l_1 \leq l_2 \leq \cdots \leq l_m$. The number of zeros is also straightforward computed.

Lemma 4: There are $\sum_{i=r}^{n-r-1} |p(i, r)|$ kernel strings with $r + 1$ zeros for all $n \geq 5$.

Proof: The proof is by induction over $n$.

The following example shows how to generate a kernel string from an ascending integer partition $p(n, m)$.

Example 2: The partitions of four in two part are $p(4, 2) = \{2 + 2, 1 + 3\}$. Since the cardinality of the set is two, this means that there are two kernel strings with three zeros and four ones. The kernel string are of the form $01^{p_1} 01^{p_2} 0$. Each element of a partition is the value of a $p_i$. So one kernel string is formed when $p_1 = p_2 = 2$ and the other kernel string is formed when $p_1 = 1$ and $p_2 = 3$ which are the kernel strings $0110110$ and $0101110$, respectively. From lemma 4, if $n = 9$ there are four kernel strings with three zeros because

$$\sum_{i=2}^{9-2-3} p(i, 2) = 4$$

$$p(i, 2) = p(i, 2)_{i=2} = p(2, 2) + p(3, 2) + (4, 2)$$

$$= 1 + 1 + 2$$

$$= 4$$

The kernel strings are shown in the following table.

<table>
<thead>
<tr>
<th>$p(2, 2)$</th>
<th>010110</th>
</tr>
</thead>
<tbody>
<tr>
<td>$p(3, 2)$</td>
<td>010110</td>
</tr>
<tr>
<td>$p(4, 2)$</td>
<td>0110110, 0101110</td>
</tr>
</tbody>
</table>

A method to process edge covers for path graphs using an efficient algorithm for generating ascending compositions of an integer $n$ in $m$ parts has been presented. Although the number of edge covers grows exponentially, the space and time required to store either the partition
diagram or the set of kernel strings is quadratic. The implication of this result is that, in sensible applications we will expeditiously recover subsets of edge covers for a given path graphs as kernel strings.
CHAPTER 2
EFFICIENT DATA STRUCTURES FOR STORING THE PARTITIONS OF INTEGERS

In this chapter we review the paper “Efficient Data Structures for Storing the Partitions of Integers” by Rung-Bin Lin, Yuan Ze University, Chung-Li, 320 Taiwan.

2.1 Introduction

In this paper, they discussed about four data structures for storing all the partitions of an integer. Data structure is used to enumerate the partitions without doing any arithmetic operation except for computing an index to an array. The enumeration can be done either with a restriction or without any restriction. Four data structures are investigated. The most efficient data structure need only store integers if n is even or if n is odd. It is created without exhaustively enumerating all the partitions. The time complexity for creating this data structure is the same as the space complexity. The complexity is low enough to make possible for storing all the partitions of an integer up to several ten thousands.

2.2 Four Data Structures

Let the set of partitions of a positive integer n be denoted by \( \Omega \). Any element \( w \in \Omega \) is denoted as \( w = (y_1, y_2, \ldots, y_k) \), where \( \sum_{i=1}^{k} y_i = n \) for \( y_i \in I, y_i > 0, i = 1, 2, \ldots, k \leq n \) and \( y_i \) is called a part of partition \( w \). The parts of a partition are not necessarily distinct, nor do they have a fixed order. However, in order to ease the representation, we assume \( y_1 \leq y_2 \leq \cdots \leq y_k \). Note that the value of \( k \) can vary from one partition to another. For example, the partitions of 6 are \( (1,1,1,1,1,1), (1,1,1,1,2), (1,1,1,3), (1,1,2,2), (1,1,4), (1,2,3), (1,5), (2,2,2), (2,4), (3,3), \) and \( (6) \). In the following we will discuss how the partitions of an integer can be stored in a computer. Four data structures are investigated. They are called direct linear,
multiplicity linear, tree, and diagram structures. The first two structures store data in a linear array, so they are called linear structures.

2.1 Linear Structures

Given the set of partitions of an integer n, said Ω {w₁, w₂, ..., wₚ}, the partitions can be stored in a one-dimensional array in the form of |w₁|w₁|w₂|w₂|......|wₚ|wₚ, where |wᵢ| denotes the number of parts in wᵢ. This is called as direct linear in this paper and it can be created by enumerating the partitions with the help of an algorithm. Once the data structure is created, it can be employed to enumerate the partitions one-by-one by first deciding the number of parts in a partition and then by retrieving the parts in sequence. This data structure is not amenable to other types of enumeration with a restriction such as enumerating partitions with the smallest part larger than a certain number.

A partition of an integer can also be represented by the repetitive parts contained in the partition. For example, the partition 〈1,1,1,3〉 can be denoted by (1,3)(3,1), where (1,3) means that part 1 occurs three times in this partition and (3,1) means that part 3 occurs only once, i.e., 1’s multiplicity is 3 and 3’s multiplicity is 1, respectively. The partitions can all be stored in an array as |w₁| (w₁ : m₁)|w₂| (w₂ : m₂)...|wₚ| (wₚ : mₚ), where |wᵢ| denotes the number of distinct parts. wᵢ : mᵢ represents all the pairs of (part, multiplicity) in partition wᵢ. This data structure is called multiplicity linear.

2.2 Tree Structure

Here they proposed a tree structure to store all the partitions of an integer. The basic idea comes from an observation that two partitions of an integer may differ in only a few parts. For example, 〈1,1,1,1,1〉 and 〈1,1,1,1,2〉 differs in only two parts. In this situation, a sequence of the branches in a tree can be used to store those parts common in any two partitions. For
example, a tree that stores all the partitions of 6 is shown in Figure 2.1. Here, a tree node is denoted by \((y, Y)\), where \(y\) is a part of a partition and \(Y\) is a number remained to be divided into parts that are at least as large as \(y\). For a root node, \(y\) is not a part and simply used to denote the least number into which \(Y\) should be divided. Prior to discussing how this tree is constructed, let us see how this tree can be used to enumerate all the partitions of 6. Starting from the root, traverse the tree in depth-first-search. When a leaf node is visited, print out the parts encountered along the path from the root to the leaf. These parts except the one stored in the root form a partition of 6 and the path length is equal to the number of parts in the partition.

A path length from the root to a leaf is defined as the number of edges traversed from the root to the leaf. A general partition tree is presented in Figure 2.2, where \(f()\) is a floor function. The pseudo code for creating such a tree is presented in Figure 2.3. Some detailed explanations of the algorithm will be presented when we elaborate on the proofs of some lemmas given later.

Figure 2.1: Tree Structure for Integer 6.
In the following some definitions and lemmas are given that are used to prove a theorem that gives the number of nodes in a partition tree.

**DEFINITION 1**: A node without any child is called a leaf node and is denoted by \((y,0)\).

**DEFINITION 2**: A node with at least one child is called an internal node and is denoted by \((y,Y)\) with \(Y>0\).
LEMMA 1: if (y,Y) is an internal node, then $0 < y \leq Y$

PROOF: It is clear that $0 < y \leq Y$ holds if (y,Y) is a root node. Initially, the queue contains only the root node (1,n). For any node (y,Y) removed from the queue, a child node (0,Y) will be created for (y,Y) and thus (y,Y) is an internal node. Based on this statement, the root node is an internal node. Furthermore, for any, $y \leq j \leq Y/2$ will create a node $(j,Y-j)$ which will also be put into the queue. The condition $y \leq j \leq Y/2$ implies that $Y-j \geq Y/2$ and $0 < y \leq j \leq Y-j$. Therefore, any node (y,Y) removed from the queue is an internal node and has the property $0 < y \leq Y$. □

LEMMA 2: Given a node $(y_p,Y_p)$ and its child node $(y_c,Y_c)$, $y_p \leq y_c$

PROOF: From lines 4, 5, 6, 7, 8, and also from Lemma 1, a child node is created only when the child’s part is greater than or equal to its parent’s part. Thus $y_p \leq y_c$. □
2.3 Diagram Structure

As it can be seen from Figure 2.1, the partitions of a node (2,4) form a sub-structure of the partitions of a node (1,4). We can in fact create a data structure for (2,4) simply using a pointer pointing to the data structure for (1,4) and remember where the shared data structure begins. In this manner we can create a data structure for the partitions of an integer without actually enumerating all the partitions. Because of the sharing of data structure, the new data structure is no longer a tree. We thus call it a partition diagram, which is in fact a directed acyclic graph. Figure 5 shows an example of sharing data structure in a partition diagram for integer 6. The total number of nodes in this partition diagram is 16. For being able to perform enumeration correctly, we must know the place where the shared data structure begins. The starting place can be stored in a node to facilitate enumeration.

To facilitate our discussion, the following definitions are given.

DEFINITION 3: A node \((y,Y)\) that has the largest \(Y\) is called an anchored node in a partition diagram.

The anchored node is also the node last added to a partition diagram. For example \((1,6)\) is the anchored node for the partition diagram of integer 6.

DEFINITION 4: A node \((y,Y)\) with \(Y=0\) is called a terminal node.
Figure 2.4: A Partition Tree with Data Structure Sharing.

An algorithm for creating such a partition diagram is given in Figure 2.3. We create a partition diagram recursively for each of the integers starting from 1 to \( n \). When we build a partition diagram for integer \( m \), i.e., a partition diagram with the anchored node \((1,m)\), we have to create a terminal node \((m,0)\) and also the internal nodes \((2, m-2), (3, m-3), \ldots, (f(m/2), (m-f(m/2))\) that store the pointers to the shared data structures. Note that node \((1, m-1)\) is created in the previous iteration. The shared data structure pointed by an internal node \((k, m-h)\) for \( 2 < h \leq f(m/2) \) is located in the partition diagram rooted at \((1, m-h)\) which has been created previously. Therefore, the sharing of data structure can be done easily. For example, given \( m=6 \), before creating node \((1,6)\) we have to create nodes \((1,5), (2,4), (3,3)\), and \((6,0)\) and the sharing is done as shown in Figure 2.4.
The enumeration can be done simply as that for a partition tree. That is, a path from the anchored node of a partition diagram to any of the terminal nodes defines a partition of integer $n$. In fact, if a partition diagram is expanded based on Lemma 2, a corresponding partition tree will be generated.
CHAPTER 3: OUR APPROACH

APPROACH

3.1 Introduction

Computer science is often difficult to define. This is probably due to the unfortunate use of the word “computer” in the name. As you are perhaps aware, computer science is not simply the study of computers. Although computers play an important supporting role as a tool in the discipline, they are just that—tools.

Computer science is the study of problems, problem-solving, and the solutions that come out of the problem-solving process. Given a problem, a computer scientist’s goal is to develop an algorithm, a step-by-step list of instructions for solving any instance of the problem that might arise. Algorithms are finite processes that if followed will solve the problem. Algorithms are solutions.

Computer science emphasizes two important topics: data structures and algorithms. Those topics are important because the choices you make for a program's data structures and algorithms affect that program's memory usage (for data structures) and CPU time (for algorithms that interact with those data structures).

This paper initiates a two-part series that explores data structures and algorithms. When choosing a data structure or algorithm, you sometimes discover an inverse relationship between memory usage and CPU time: the less memory a data structure uses, the more CPU time associated algorithms need to process the data structure’s data items, which are primitive type values or objects, via references. Also, the more memory a data structure uses, the less CPU time associated algorithms need to process the data items—and faster algorithms result. This
paper begins with a presentation of basic concepts and continues with a tour of the array data structure.

3.2 Approach

Consider ‘m’ objects which have to be divided into ‘n’ bins in such a way that the next bin should not contain less objects than the previous bin. Suppose first bin contains 1 object then all other bins should contain at least 1 object. Similarly if the fourth bin contains 3 objects then all other bins after fourth bin should contain a minimum of 3 objects. This is the method of dividing objects in the form of steps into the bins.

The number of ways of dividing m objects in n bins is represented as \( f(m, n) \).

The number of objects that are being divided into the bins will remain same or in increasing order which is in the form of steps but never decreases.

3.3 Method

Let there be ‘m’ distinct objects say 1o, 2o, 3o, 4o……mo and ‘n’ distinct bins say 1b, 2b, 3b……nb. Let the function of dividing ‘m’ objects into ‘n’ bins be \( f(m, n) \). The ‘m’ objects should be divided in ‘n’ bins by satisfying the following conditions:

1. Each bin can contain any number of objects.

2. The objects should be divided in such a way so that the successor bin should always have more than or equal number of objects than its predecessor bin.

Example: Consider x contains y objects then \((x+1)\) should contain \( \geq y \) objects.
Example

Consider an example of dividing 8 objects in 3 bins. The objects can be divided in the following ways:

<table>
<thead>
<tr>
<th>b1</th>
<th>b2</th>
<th>b3</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>8</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>7</td>
</tr>
<tr>
<td>0</td>
<td>2</td>
<td>6</td>
</tr>
<tr>
<td>0</td>
<td>3</td>
<td>5</td>
</tr>
<tr>
<td>0</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>6</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td>5</td>
</tr>
<tr>
<td>1</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>4</td>
</tr>
<tr>
<td>2</td>
<td>3</td>
<td>3</td>
</tr>
</tbody>
</table>

In the above example if the bin 0 is filled with one object then all other should be filled with a minimum of 0 object and not less than that. Then the function for remaining objects is represented as \( f(8, 3) \) which there are 10 ways that can be filled in 3 bins.

Similarly if first bin is filled with 1 object then all other bins should be filled with a minimum of one object. The dividing process goes by following this condition till the last bin is filled with the last object.
3.4 Generating an Algorithm for Dividing Objects in Non-descending Order

Partitioning an integer \( n \) is to divide it into its constituent parts which are all positive integers. Algorithms for enumerating all the partitions of an integer or only the partitions with a restriction have long been invented [1,2].

Consider \( f(8,3) \) i.e dividing 8 objects into 3 bins. If bin1 contains 0 objects then the function for dividing the other objects is \( f(8,2) \) which means 8 objects should be divided in 2 bins. Similarly if bin1 contains 1 object then the function for dividing the other objects is \( f(5,2) \) which means 5 objects should be divided in 2 bins because if first bin is filled with one object then other two bins should also be filled with a minimum of one object. So the remaining objects to be filled are 8-3= 5. Similarly if bin1 and bin2 contains 2 object then the function for dividing the other objects is \( f(2,1) \) which means 1 objects should be divided in 1 bin because if first and second bean is filled with two objects then other beans should also be filled with a minimum of two objects. So the remaining objects to be filled are 8-6= 2.

Let \( f(m,n) \) be the number of ways of dividing \( m \) objects into \( n \) bins with non-descending cardinality. Function \( f(m,n) \) for dividing \( m \) objects into \( n \) bins in this particular format is shown

\[
f(m,n) = f(m, n-1) + f(m-n, n-1) + f(m-2n, n-1) + f(m-3n, n-1) + \ldots + f(m- \lfloor m/n \rfloor * n, n-1)
\]

// first bin contains 0 objects

// first bin contains 1 object

// first bin contains 2 objects

// first bin contains 3 objects

// \ldots

// first bin contains \( \lfloor m/n \rfloor \) objects.
3.5 Partition Diagram

Algorithms for enumerating all the partitions of an integer or only the partitions with a restriction have been extensively studied [1], [5].

The diagram represents a directed acyclic graph.

A data structure called partition diagram for storing all the partitions of an integer is proposed in [1]. In Merca [4], [5] improvements are proposed which, to date, are the most adequate data structures for generating integer partitions. We use the data structure proposed by Merca to present an efficient algorithm for generating ascending compositions of an integer n in m parts.
The partition diagram is a directed acyclic graph. Anode in the partition diagram is denoted by \((m,n)\) where \(m\) is the number of objects and \(n\) denotes the number of bins. A node \((m,n)\) that has no predecessor is called an anchored node (root node) in a partition diagram. A node \((m,n)\) which has no successor is called a terminal node. For example in the Fig. 1 the node \((17,6)\) is an anchored node and also internal node, whereas node \((2,1)\) is a terminal node (leaf node).

Given a partition diagram, a path from an anchored node to terminal node defines a unique partition in which \(m\) objects are divided into \(n\) bins.

For example in Fig. 1 the path \((17, 6)\) \((17, 5)\) \((12, 4)\) \((4, 3)\) \((4, 2)\) defines a partition.

If the number of objects in the first bin is ‘\(a\)’ then all bins should have at least ‘\(a\)’ objects. If we allocate ‘\(a\)’ objects to every bin then we have \((m-na)\) objects left to be distributed in \((n-1)\) bins. So the process of dividing should continue till the value of \(m-n\) is greater than 0.

When we format the algorithm we can assume two situations

- **The first bin is empty**

  In this case \(m\) objects are to be distributed in \((n-1)\) bins.

- **The first bin contains at least one object**

  In this case we allocate one object to every bin. Thus we have \((m-n)\) objects left to be distributed in \(n\) bins.
Algorithm

function f(m, n)
{
    if (m ≥ n)
    {
        if (n ≡ 1) return 1;
        else return (f (m, n-1) +f (m-n, n));
    }
    else return f(m, n-1);
}

Figure 3.2: Algorithm for Non-Descending Partition.

Lemma 1:  \( f(m, n) \leq xf(m-1, n) \leq nf(m-1, n) \), where \( x \) is the number of steps explained below.

Consider \( m \) objects to be placed into \( n \) bins. Let \( x \) be the number of steps that are formed while arranging the objects.

Figure 3.3: Different ways of Arranging the Objects
If the successor bin contains more objects than the preceding bin then the objects are arranged in the form of steps in between the bins. For example in Fig 3.2 shows a configuration in which bin 2 contains more number of objects than bin 1 so they are arranged as a step. Similarly bin 2 and bin 3, bin 4 and bin 5, bin 7 and bin 8, bin 8 and bin 9. If the successor bin contains the same number of objects as in the preceding bin then the arrangement is not formed as step. It remains at same level as they contain same number of objects. For example bin 3 and bin 4 contains same number of objects so they remain at same level. Similarly bin 5, bin 6 and bin 7 contain same number of objects.

Figure 3.4: Different ways of Adding an Object in the Above Arrangement

If we want to add an object in the configuration shown in Fig 3.2, the added object should not affect the configuration i.e. the object is to be added in non-descending cardinality. As shown in Fig 3.3, the added object can only be placed in the shaded portion. Therefore we can say that the number of arranging the objects in the predecessor bin is always less than or equal to the number of way of arranging the objects in the successor bin in the form of steps. When we have a configuration for $f(m-1, n)$ then we can generate at most $xf(m-1, n)$ configurations when we add an object, where ‘x’ is the number of steps in the configuration for $f(m-1, n)$. Hence $f(m,n) \leq xf(m-1,n)$ . Since $x \leq n$ we can say $f(m,n) \leq xf(m-1,n) \leq nf(m-1,n)$. □
Lemma 2: \( f(m, n) \leq \sqrt{2m} f(m-1, n) \)

From the above lemma we know that \( f(m, n) \leq xf(m-1, n) \) where \( x \) is the number of steps.

![Image](image.png)

Figure 3.5: Arrangement of Objects to Achieve Maximum Steps

Maximum number of steps can be obtained by placing the objects in the following way.

Let there be many number of objects with many number of bins. Then the objects are placed in increasing way to obtain the maximum steps in the following way

- 0 objects in bin 1
- 1 object in bin 2
- 2 objects in bin 3
- 3 objects in bin 4
- ...
- \( t \) objects in bin \( t+1 \).

Let \( m \) be the total number of objects, then

\[
m = \left\lfloor \frac{t(t+1)}{2} \right\rfloor / 2
\]

By solving this we get \( t = \sqrt{2m} \)
The maximum number of steps is $\sqrt{2m}$

Therefore $f(m,n) \leq \sqrt{2m} f(m-1,n)$

Algorithms which efficiently built these kind of integer partition combinations have long been studied, a survey can be found in Knuth [8]. Although the space and time needed to store either the partition diagram or the set of kernel strings is quadratic [9], our approach creates the most efficient data structure with space and time complexity $O(mn)$. The space and time complexity is low enough to make possible for storing all the partitions of an integer up to several ten thousands. In [1] $O(n^2)$ storage is used for storing the partitions. Our algorithm uses less space when $m$ is smaller than $n$. The implication of this result is that, in practical applications, we can efficiently recover subsets for a given path graph as kernel strings are the base to generate combined strings.
Consider a case in which $x_0$ bin contains 0 objects, $x_1$ bin contains 1 object, $x_2$ bin contains 2 objects, $x_3$ bin contains 3 objects …… $x_a$ bin contains ‘$a$’ objects.

Let there be ‘$m$’ objects. Then we can say $x_0 + x_1 + x_2 + x_3 + x_4 + … + x_a = m$

Example: Consider an example of dividing 4 objects into 3 bins such that $x_1 + x_2 + x_3 = m$

The only case we have is

$x_1 = 1, x_2 = 0, x_3 = 1$

This method of partitioning the objects contains less number of ways than the method of partitioning objects in traditional method i.e $x_0 + x_1 + x_2 + x_3 + x_4 + … + x_a = m$.

Now consider the same example of dividing 4 objects into 3 bins such that $c_1 + c_2 + c_3 = m$

The possible cases are follows

$c_1 = 0, c_2 = 0, c_3 = 4$
$c_1 = 0, c_2 = 1, c_3 = 3$
$c_1 = 0, c_2 = 2, c_3 = 2$
$c_1 = 0, c_2 = 3, c_3 = 1$
$c_1 = 0, c_2 = 4, c_3 = 0$
$c_1 = 1, c_2 = 1, c_3 = 2$
$c_1 = 1, c_2 = 2, c_3 = 1$
$c_1 = 1, c_2 = 0, c_3 = 3$
$c_1 = 1, c_2 = 3, c_3 = 0$
$c_1 = 2, c_2 = 1, c_3 = 1$
\[ c_1 = 2, c_2 = 0, c_3 = 2 \]
\[ c_1 = 2, c_2 = 2, c_3 = 0 \]
\[ c_1 = 3, c_2 = 0, c_3 = 1 \]
\[ c_1 = 3, c_2 = 1, c_3 = 0 \]
\[ c_1 = 4, c_2 = 0, c_3 = 0 \]

4.2 Generating an Algorithm for Partitioning Objects

Initially consider there are \( m+n-1 \) positions out of which choose \( n-1 \) positions then you will be left with \( m \) objects to be partitioned into \( n \) bins.

Example:

Consider the following 4 objects to be partitioned into 3 bins.

1. Initially consider \( m+n-1 \) positions i.e. \( 4+3-1=6 \)

   \[ \square - \square - \square - \square - \square - \square \]

2. Choose \( n-1 \) positions i.e. \( 3-1=2 \). There are \( \binom{6}{2} \) ways. The two positions can be taken in the following ways.

   \[ \blacksquare - \blacksquare - \square - \square - \square - \square \]

   Bin1 contains 0 objects

   Bin2 contains 0 objects

   Bin3 contains 4 objects

   \[ \blacksquare - \square - \blacksquare - \square - \square - \square \]

   Bin1 contains 0 objects
Bin2 contains 1 objects
Bin3 contains 3 objects

Bin1 contains 0 objects
Bin2 contains 2 objects
Bin3 contains 2 objects

Bin1 contains 0 objects
Bin2 contains 3 objects
Bin3 contains 1 objects

Bin1 contains 0 objects
Bin2 contains 4 objects
Bin3 contains 0 objects
Bin1 contains 1 objects
Bin2 contains 0 objects
Bin3 contains 3 objects

Bin1 contains 1 objects
Bin2 contains 1 objects
Bin3 contains 2 objects

Bin1 contains 1 objects
Bin2 contains 2 objects
Bin3 contains 1 objects

Bin1 contains 1 objects
Bin2 contains 3 objects
Bin3 contains 0 objects
Bin1 contains 2 objects
Bin2 contains 0 objects
Bin3 contains 2 objects

Bin1 contains 2 objects
Bin2 contains 1 objects
Bin3 contains 1 objects

Bin1 contains 2 objects
Bin2 contains 2 objects
Bin3 contains 0 objects

Bin1 contains 3 objects
Bin2 contains 1 objects
Bin3 contains 0 objects
Consider partitioning m objects into n bins with m ≥ n.

Now we consider partition of ‘m’ objects into bins such that a bin can contain at most ‘a’ objects.

The formula for this is \( 0x_0 + 1x_1 + 2x_2 + 3x_3 + \ldots \ldots \ldots ax_a = m \) which is less than the number of ways m can be partitioned into a bins by \( x_0 + x_1 + x_2 + x_3 + \ldots \ldots \ldots x_a = m \) which is

\[
\leq \binom{m + a}{a}
\]

(1)

Now consider the ways m objects can be partitioned into bins in such a way that each bin contains at least a objects. The number of bins is now restricted by \( n/(a+1) \).

We first choose a position among n bins with n ways

(2)

Let the position be \( p \). Let each of bin 1 to bin \( p \) contains \( \leq a \) objects and each of bin \( p + 1 \) to bin \( n \) contains \( > a \) objects.
For the first part (each of bin 1 to bin \( p \) contains \( \leq a \) objects) the number of ways is bounded by (1)

For the second part (each of bin \( p+1 \) to bin \( n \) contains at least \( a \) objects) there are no more than \( n/a \) bins, thus there are no more than ways of partitioning \( m \) objects into \( m/(a+1) \) bins i.e

The number of ways is bounded by

\[
\binom{m + (m/(a + 1)) - 1}{m/(a + 1) - 1}
\]

(3)

So the total number of ways of partitioning '\( m \)' objects into '\( n \)' bins is less than the product of (1), (2) and (3), that is

\[
\leq \binom{m + a}{a} \cdot \binom{m + (m/(a + 1)) - 1}{n/(m + 1) - 1}
\]

\[
\approx (m/a)^a n(a)^{m/a}
\]

Therefore we let

\[
(m/a)^a = (a)^{m/a}
\]

We have \( a = \sqrt{m} \)

This gives about \( (\sqrt{m})^{\sqrt{m}} \) ways, while the number of ways of partitioning objects using the formula \( x_1 + x_2 + x_3 + x_4 \ldots + x_n = m \) gives

\[
\binom{m + n - 1}{n - 1} \approx (m/n)^n
\]
CONCLUSION

In this thesis we have studied about partitioning a set into non-descending cardinality. The space and time complexity for creating a linear structure or a partition tree is proportional to the number of partitions whereas the complexity for creating a partition diagram is only $O(mn)$. This complexity allows us to create a partition diagram that can store all the partitions of an integer up to several ten thousands.
REFERENCES


VITA

Oshani Titti was born on July 09, 1992, in Andhra Pradesh, India. She completed her schooling in Andhra Pradesh and graduated from high school in 2007. She then completed her Bachelor’s degree in Computer Science Engineering from Anil Neerukonda Institute of Technology and Science, affiliated to Andhra University in 2013.

She worked for Tata Consultancy and Services, Bangalore, India as Assistant Software Engineer from June 2013 to July 2014.

In fall 2014, she came to US for her masters and took several courses towards Software Engineering.