Recursive Algorithm for Generating Partitions of an Integer

Sung-Hyuk Cha

Computer Science Department, Pace University
1 Pace Plaza, New York, NY 10038 USA
scha@pace.edu

Abstract. *This article first reviews the problem of partitioning a positive integer in general. And then recursive algorithms for $P(n,k)$, generating exactly k partitions of a positive number n are presented. The most efficient recursive algorithm's computational running time is $\Theta(k \times p(n,k))$ which is the lower bound of $P(n,k)$ problem.*

1 Preliminary

A positive integer n can be represented by sums of positive integer terms. For example of $n = 5$, there are 7 different ways as listed in Table 1.

Table 1. $P(5)$ and $P(5,k)$ example

5	= 1 + 1 + 1 + 1 + 1	$P(5,5)$ = {<1,1,1,1,1>}
	= 2 + 1 + 1 + 1	$P(5,4)$ = {<2,1,1,1>}
	= 3 + 1 + 1 = 2 + 2 + 1	$P(5,3)$ = {<2,2,1>, <3,1,1>}
	= 4 + 1 = 3 + 2	$P(5,2)$ = {<3,2>, <4,1>}
	= 5	$P(5,1)$ = {<5>}

This classic problem of generating partitions of a positive integer n is formally defined in eqn(1)

$$P(n) = \{l \mid \sum_{l(i) \in l} l(i) = n \wedge l(i) \in \text{integer}\{1,...,n\} \wedge l(i) \geq l(j) \text{ if } i > j\} \qquad (1)$$

Let $l(i)$ denotes the i^{th} element in the list l. $P(n)$ is a set of ordered list l whose elements $l(i)$'s are positive integers and their sum is exactly n. Any permutation of l is considered to be the same, e.g., (2 + 2 + 1) = (1 + 2 + 2) = (2 + 1 + 2). Hence, only the sorted lists are included in the output set.

A simple binary tree to generate all partitions of all integers is depicted in [1] and Figure 1 (a). The algorithm is given below where $+(l_x, l_y)$ is concatenating two lists and $l(1 \sim |l|-1)$ is a sub list of l starting from the fist element to $|l|-1^{th}$ element, e.g., +(<3,3>,<1>) = <3,3,1> = l_x and $l_x(1 \sim 2)$ = <3,3>.

Algorithm 0: $P(*)$ where

1. the root is <1>
2. generate children by eqns (2) and (3) for all nodes, recursively.

$$L(l) = +(l, <1>) \qquad (2)$$

$$R(l) = \begin{cases} +(l(1 \sim |l|-1), <l(|l|)+1>) & \text{if } l(|l|) < l(|l-1|) \\ \text{abort} & \text{otherwise} \end{cases} \qquad (3)$$

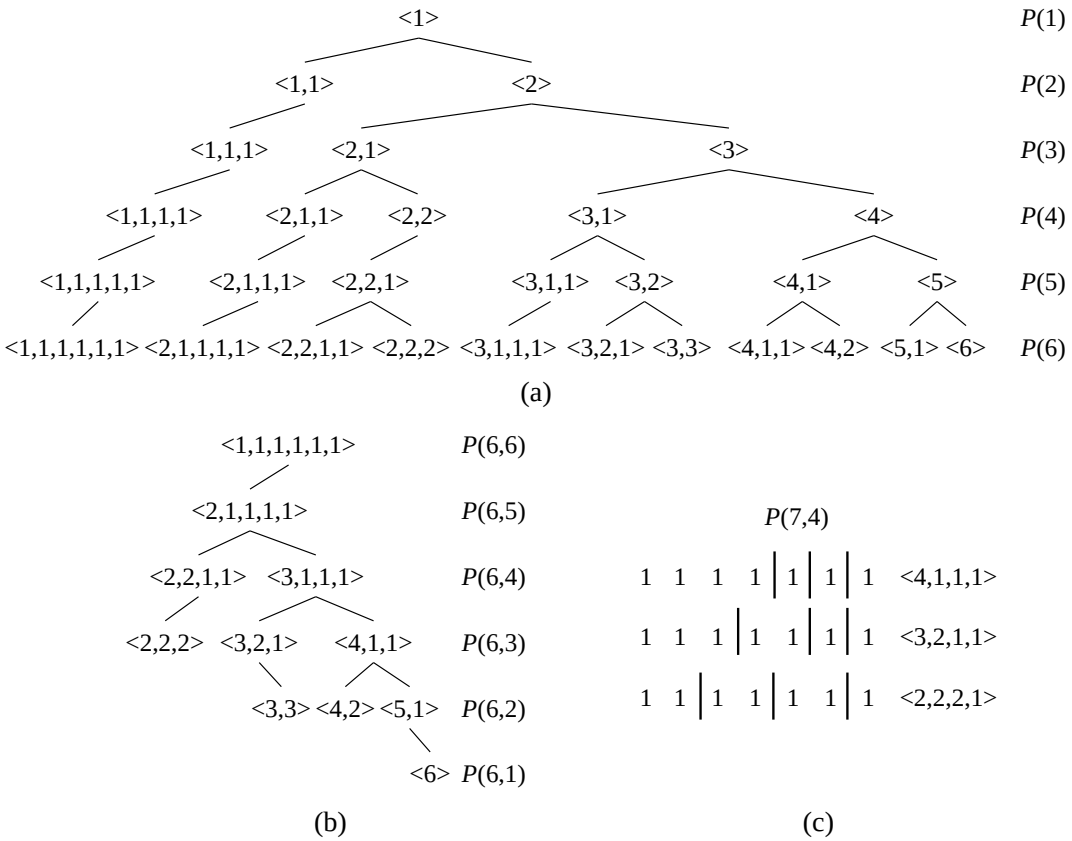


Figure 1. (a) a binary tree for $P(*)$, (b) a binary tree for $P(6)$, and (c) $P(7,4)$ with bars.

$P(n)$ for any n can be generated together with all $P(i)$'s for $i < n$ using the Algorithm 0, yet another more efficient algorithm to generate only $P(n)$ is depicted as a simple binary tree in [1] and an example is shown in Figure 1 (b).

Let $P(n,k)$ denote a set of ordered list l whose length is exactly k, i.e., the positive integer n is expressed exactly k partitions as formally defined in eqn(4) and some examples are given in Table 2.

$$P(n,k) = \{l = < l(1), \cdots, l(k) > | \sum_{i=1}^{k} l(i) = n \wedge l(i) \in \text{integer}\{1,...,n\} \wedge l(i) \geq l(j) \text{ if } i > j\} \quad (4)$$

$$P(n) = \bigcup_{k=1}^{n} P(n,k) \quad (5)$$

Notice that k^{th} level in the binary tree in Figure 1 (b) contains $P(n,k)$ and thus $P(n,k)$ can be computed using the respective algorithm, yet one must generate $P(n,1)$~$P(n,k-1)$ in order to generate $P(n,k)$. Another way to think of the problem is using $k - 1$ bars to separate n items into k partitions.

Table 2. Examples of $P(n,k)$

$P(15,2)$	$P(9,3)$	$P(9,4)$	$P(9,5)$	$P(8,3)$	$P(8,4)$	$P(8,5)$	$P(7,3)$	$P(7,4)$
<8,7>	<3,3,3>	<3,2,2,2>	<2,2,2,2,1>	<3,3,2>	<2,2,2,2>	<2,2,2,1,1>	<3,2,2>	<2,2,2,1>
<9,6>	<4,3,2>	<3,3,2,1>	<3,2,2,1,1>	<4,2,2>	<3,2,2,1>	<3,2,1,1,1>	<3,3,1>	<3,2,1,1>
<10,5>	<4,4,1>	<4,2,2,1>	<3,3,1,1,1>	<4,3,1>	<3,3,1,1>	<4,1,1,1,1>	<4,2,1>	<4,1,1,1>
<11,4>	<5,2,2>	<4,3,1,1>	<4,2,1,1,1>	<5,2,1>	<4,2,1,1>		<5,1,1>	
<12,3>	<5,3,1>	<5,2,1,1>	<5,1,1,1,1>	<6,1,1>	<5,1,1,1>			
<13,2>	<6,2,1>	<6,1,1,1>						
<14,1>	<7,1,1>							

Generating $P(n,k)$ is not only an important *number theory* problem, but also plays a key role in numerous application problems. Albeit an efficient was developed in 1779 and described in [1], here recursive algorithms are presented not only to provide a recursive definition of the problem, but also to reason its efficiency.

2 Designing Recursive Algorithms

Before embarking on the algorithm, it is necessary to define some concatenate functions. The function, $+(x,L)$, takes an integer x and L, a set of ordered lists of integers as inputs as defined in eqn(6). A new set of lists are produced by inserting x in the head of each list in L as exemplified in eqn(7). Inserting an element in the beginning of a list takes a constant computational time [2].

$$+(x,L)=\{<x,l(1),\cdots,l(|\,l\,|)>|\,l\in L\} \tag{6}$$

$$\begin{Bmatrix}<4,6,1,1>\\<4,5,2,1>\\<4,4,3,1>\\<4,4,2,2>\\<4,3,3,2>\end{Bmatrix}=+(4,\begin{Bmatrix}<6,1,1>\\<5,2,1>\\<4,3,1>\\<4,2,2>\\<3,3,2>\end{Bmatrix}) \tag{7}$$

Some of the lists in the output set in eqn(6) may not be ordered lists anymore. Hence, let's define a new function, $\oplus(x,L)$ which excludes such cases as defined in eqn(8).

$$\oplus(x,L)=\{<x,l(1),\cdots,l(|\,l\,|)>|\,l\in L\wedge l(1)\le x\} \tag{8}$$

$$\begin{Bmatrix}<4,4,3,1>\\<4,4,2,2>\\<4,3,3,2>\end{Bmatrix}=\oplus(4,\begin{Bmatrix}<6,1,1>\\<5,2,1>\\<4,3,1>\\<4,2,2>\\<3,3,2>\end{Bmatrix}) \tag{9}$$

$$\{\}=\oplus(1,\begin{Bmatrix}<3,1,1>\\<2,2,1>\end{Bmatrix}) \tag{10}$$

The output of $\oplus$ may be an empty set as in eqn(10). Note that $\oplus(4,\{\})$ returns <4>. Let $\otimes(x,L)$ be identical to $\oplus(x,L)$ except that $\otimes(4,\{\})$ returns $\{\}$ instead of <4>.

$$\otimes(x,L)=\begin{cases}\{\} & \text{if } L=\{\}\\ \{<x,l(1),\cdots,l(|\,l\,|)>|\,l\in L\wedge l(1)\le x\} & \text{otherwise}\end{cases} \tag{11}$$

Using the $\otimes(x,L)$ function, $P(n,k)$ = $\{\oplus(1,\ P(n-1,k-1),\ \oplus(2,\ P(n-2,k-1),\ \ldots,\ \oplus(n,\ P(n-n,k-1))\}$ and thus, a recursive and algorithmic definition of $P(n,k)$ can be written as in eqn(12).

Algorithm I: eqn(12)

$$P(n,k)=\begin{cases}\{\} & \text{if } k>n\\ n & \text{if } k=1\,\&\,k\le n\\ \bigcup_{i=1}^{n}\otimes(i,P(n-i,k-1)) & \text{if } k>1\,\&\,k\le n\end{cases} \tag{12}$$

One should not generate the *Fibonacci sequence number* by its recursive definition as it would take $\Theta(\varphi^n)$. Similarly, the eqn(12) is a sound recursive definition of $P(n,k)$ but unfortunately Algorithm I makes too many unnecessary recursive procedure calls as shown in Figure 2 (a).

One quick observation from Figure 2 (a), $n-k+1$ upper bound can be used to reduce the unnecessary recursive calls.

Theorem 1. $P(n-i,k-1) = \{\}$ if $i > n-k+1$.
Proof. If $i = n-k+1+\varepsilon$ where $\varepsilon \geq 1$, $P(n-(n-k+1+\varepsilon), k-1) = P(k-1-\varepsilon, k-1)$.
Since $(k-1-\varepsilon) < (k-1)$, $P(n-i,k-1) = \{\}$. ■

Instead of checking all integers from 1 to n as in Algorithm I, only from 1 to $n-k+1$ can be checked. Using the Theorem 1, Algorithm II is given in eqn (13)

Algorithm II: using $n-k+1$ upper bound in eqn(13)

$$P(n,k) = \begin{cases} n & \text{if } k = 1 \\ \bigcup_{i=1}^{n-k+1} \oplus (i, P(n-i,k-1)) & \text{if } k > 1 \end{cases} \tag{13}$$

Notice that $\oplus$ is used instead of $\otimes$ since the upper bound guarantees that $P(n,k)$ never returns an empty set because n is always greater than or equal to k. The recursive call diagram of Algorithm II for $P(6,4)$ is shown in Figure 2 (b). Once again, the eqn(13) may be a better definition for $P(n,k)$ than the eqn(4) which is a non-algorithmic definition, yet Algorithm II still makes too many unnecessary recursive calls.

Another upper bound can be noticed from Figure 2(b); $P(3,2)$ in $\oplus(1,P(3,2))$ makes two recursive calls: $\oplus(1,P(2,1))$ and $\oplus(2,P(1,1))$. Notice that $\oplus(1, \oplus(2,P(1,1)))$ will return always an empty set because the output list $<\ldots,1,2,\ldots>$ is not ordered.

Theorem 2. $\oplus(s, \oplus(j, P(n-j,k-1)) = \{\}$ if $j > s$
Proof. Let $j = s + \varepsilon$ where $\varepsilon \geq 1$. $+(s, +(s+\varepsilon, P(n-(c+\varepsilon), k-1))$ will return $< c, c+\varepsilon, \ldots>$ which is not an ordered list. Hence, $\oplus(c, \oplus(c+\varepsilon, P(n-(c+\varepsilon), k-1)) = \{\}$ ■

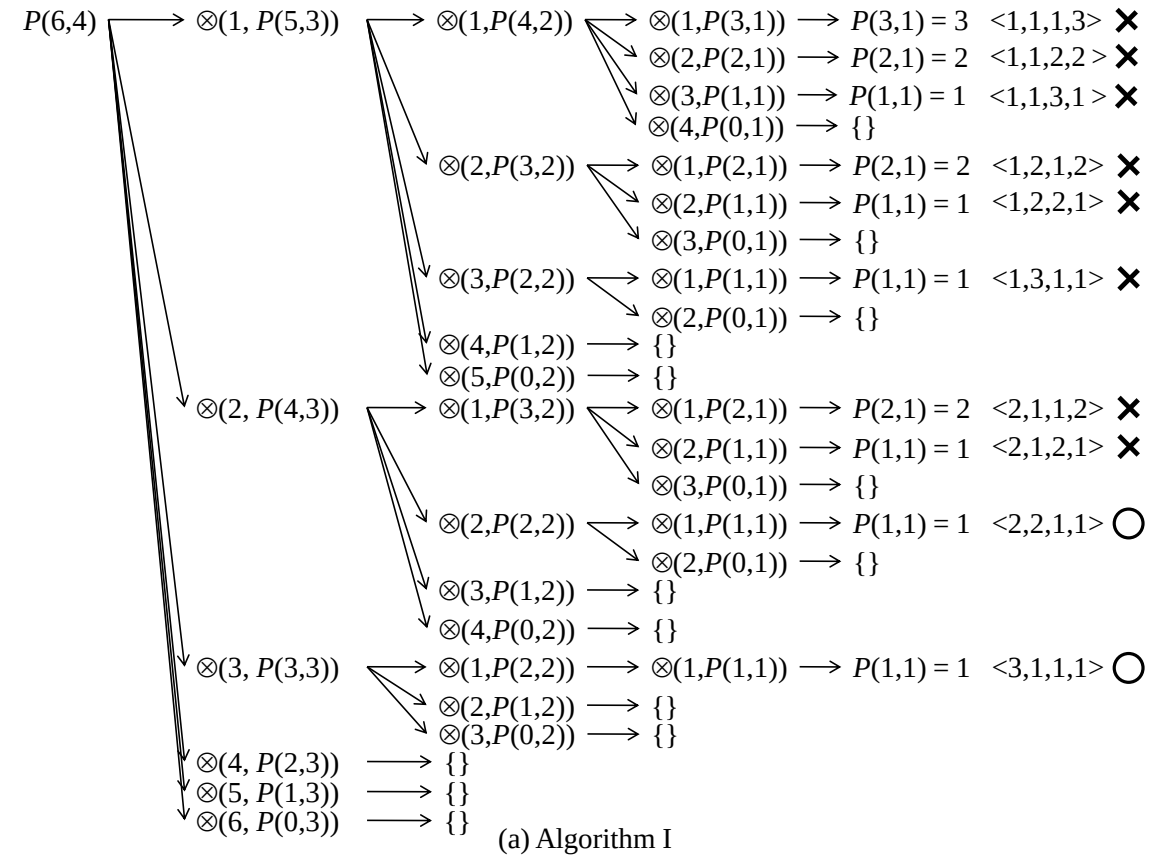
The value s in the parent level can be another upper bound for a given recursive procedure to avoid unnecessary children procedures. The value c must be passed to its children recursive procedures as an extra parameter and the recursive procedure must choose the minimum of two upper bound values. Hence, the following Algorithm III has two parts: one is the initial call part in eqn(14) and the other is recursive procedure with an extra parameter part in eqn(15).

Algorithm III: using the minimum of two upper bounds: eqn(14) followed by eqn(15)

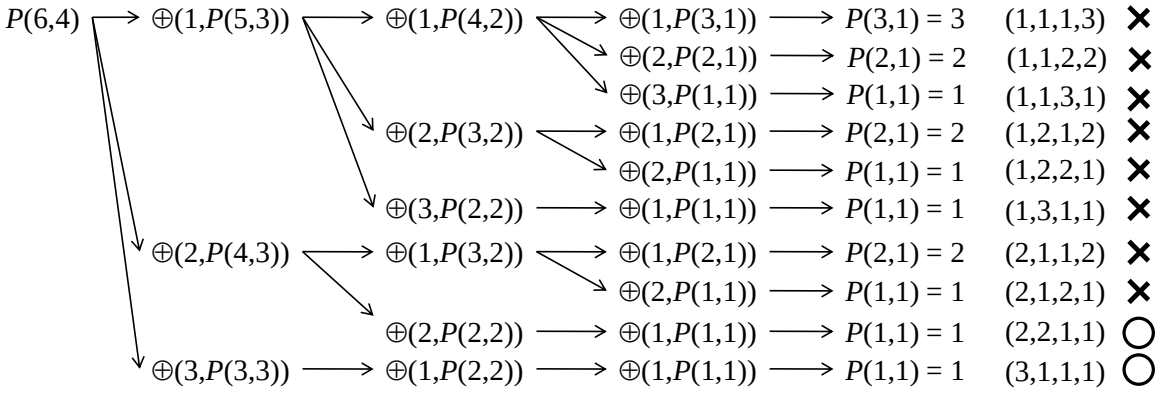
$$P(n,k) = \begin{cases} \{\} & \text{if } k > n \\ Q(n,k,n-k+1) & \text{otherwise} \end{cases} \tag{14}$$

$$Q(n,k,s) = \begin{cases} n & \text{if } k = 1 \\ \bigcup_{i=1}^{s} \oplus (i, Q(n-i,k-1,\min(i,n-i-k+2))) & \text{if } k > 1 \end{cases} \tag{15}$$

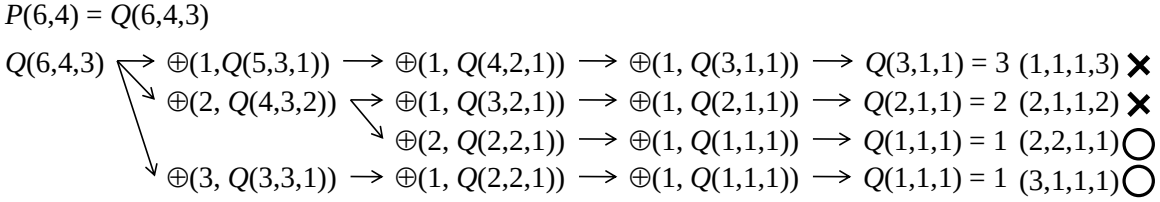
Figure 2 (c) shows the recursive call diagram of Algorithm III for $P(6,4)$. It is much more efficient than Algorithm I and II yet can be improved further.



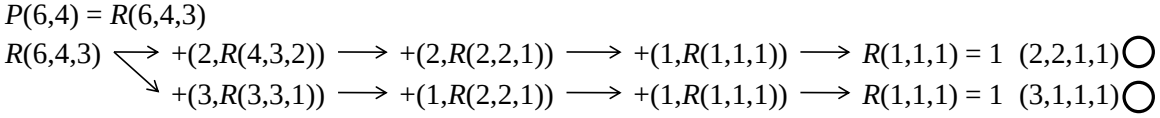
(a) Algorithm I



(b) Algorithm II



(c) Algorithm III



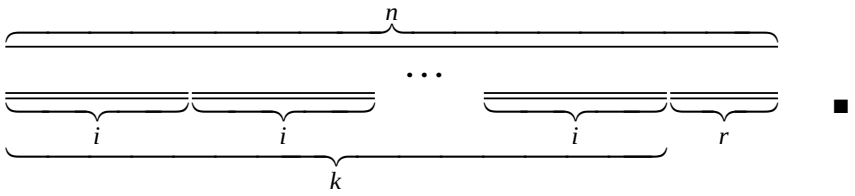
(d) Algorithm IV

Figure 2. recursive procedure call diagrams for $P(6,4)$.

So far, only the upper bound values have been utilized in Algorithm II and III, a lower bound, ceil(n/k) can be observed from Figure 2. In order to cover n with k partitions, the first partition's size must be at least ceil(n/k).

Theorem 3. $\oplus(i, P(n-i,k-1)) = \{\}$ if $i < \lceil n/k \rceil$

Proof. Suppose the first partition's size is i < ceil(n/k). Then the remaining partitions' size must be less than or equal to i. If all partitions' sizes are equal to i, the total size is the maximum and it is $k \times i$. But $k \times i < n$ because ceil(n/k).



■

The following Algorithm IV utilizes both upper bound and lower bound.

Algorithm IV: Algorithm III + lower bound: eqn(16) followed by eqn(17)

$$P(n,k) = \begin{cases} \{\} & \text{if } k > n \\ R(n,k,n-k+1) & \text{otherwise} \end{cases} \tag{16}$$

$$R(n,k,s) = \begin{cases} n & \text{if } k = 1 \\ \bigcup_{i=\lceil n/k \rceil}^{s} +(i, R(n-i,k-1,\min(i,n-i-k+2))) & \text{if } k > 1 \end{cases} \tag{17}$$

Figure 2(d) and Figure 3 show the recursive procedure call diagram of Algorithm IV. Note that $+(x,L)$ function is used instead $\oplus$ because the lower bound guarantees that all $l(1)$'s are less than or equal to x.

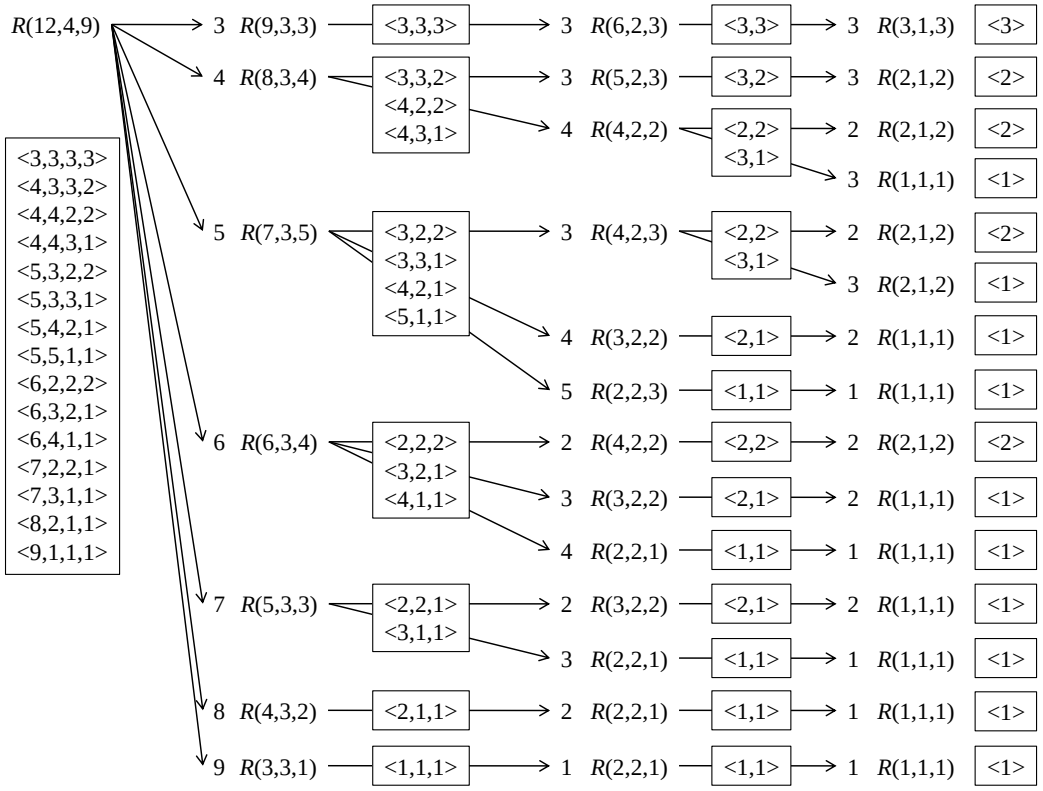


Figure 3. The recursive procedure call diagram of Algorithm IV for $P(12,4)$.

3 Computational Complexity

$p(n)$ is widely known as the *integer partition function* which is a way of writing n as a sum of positive integers [3,4]. In other words, $p(n)$ is the cardinality of $P(n)$; $p(n) = |P(n)|$. Let $p(n,k) = |P(n,k)|$. Finding $p(n)$ or $p(n,k)$ such as $p(27,8) = 352$, $p(31,5) = 427$, and some examples in Table 3 is an important problem and has been widely studied in depth. It can be computed much faster without computing $P(n,k)$ [3,4]. We will not consider the algorithm for generating $p(n)$ here but use it to state the computational complexity of Algorithm IV.

$$p(n) = \sum_{k=1}^{n} p(n,k) \tag{18}$$

Table 3. Cardinality $p(n,k)$

$n\backslash k$	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	$p(n)$
1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1
2	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	2
3	1	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	3
4	1	2	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	5
5	1	2	2	1	1	0	0	0	0	0	0	0	0	0	0	0	0	7
6	1	3	3	2	1	1	0	0	0	0	0	0	0	0	0	0	0	11
7	1	3	4	3	2	1	1	0	0	0	0	0	0	0	0	0	0	15
8	1	4	5	5	3	2	1	1	0	0	0	0	0	0	0	0	0	22
9	1	4	7	6	5	3	2	1	1	0	0	0	0	0	0	0	0	30
10	1	5	8	9	7	5	3	2	1	1	0	0	0	0	0	0	0	42
11	1	5	10	11	10	7	5	3	2	1	1	0	0	0	0	0	0	56
12	1	6	12	15	13	11	7	5	3	2	1	1	0	0	0	0	0	77
13	1	6	14	18	18	14	11	7	5	3	2	1	1	0	0	0	0	101
14	1	7	16	23	23	20	15	11	7	5	3	2	1	1	0	0	0	135
15	1	7	19	27	30	26	21	15	11	7	5	3	2	1	1	0	0	176
16	1	8	21	34	37	35	28	22	15	11	7	5	3	2	1	1	0	231
17	1	8	24	39	47	44	38	29	22	15	11	7	5	3	2	1	1	297

The output set size of $P(n,k)$ is $k \times p(n,k)$ and hence the computational running time complexity of any algorithm is $\Omega(k \times p(n,k))$ which is the lower bound. Note that Algorithm IV makes exactly $p(n,k)$ number of concatenate functions for each level. There are k levels and thus Algorithm IV takes $\Theta(k \times p(n,k))$. All Algorithms I, II, and III takes $\Omega(k \times p(n,k))$.

It should be noted that $p(n,k)$ grows very fast as depicted in Figure 4. Generating $P(n)$ takes exponential, $O(n \times p(n))$ by the eqn(19), a.k.a. the *Hardy and Ramanujan asymptotic formula* [3,4].

$$p(n) \approx \frac{1}{4n\sqrt{3}} e^{\pi\sqrt{\frac{2n}{3}}} \tag{19}$$

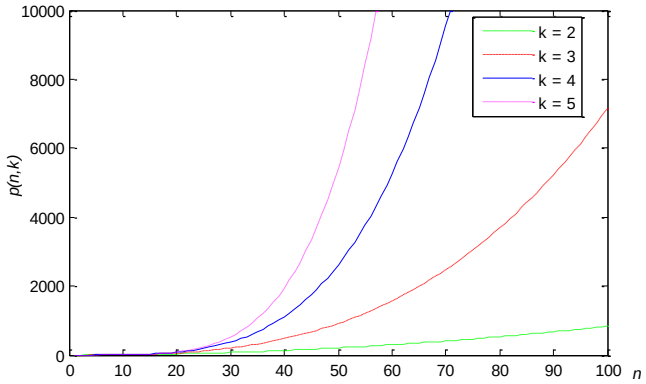


Figure 4. $p(n,2)$, $p(n,3)$, $p(n,4)$, $p(n,5)$ plots

4 Discussion

Recursive thinking often sheds some light on how to define and solve a difficult problem. This paper presented four recursive algorithms for $P(n,k)$. Naïve recursive algorithms are introduced to devise an excellent recursive algorithm. It is often a good strategy and tactic to design an algorithm for a difficult problem.

$P(n,k)$ is about distributing n unlabeled tasks to exactly k unlabeled processors. No idle processor is allowed. If processors are labeled, the order matters, e.g., <2 ,2 ,1> ≠ <2 ,1 ,2> ≠ <1 ,2 ,2>. Let $O(n,k)$ be a way of distributing n unlabeled tasks to exactly k labeled processors. To generate $O(n,k)$, Algorithm II can be used by replacing $\oplus$ with + function.

Algorithm V:

$$O(n,k) = \begin{cases} n & \text{if } k = 1 \\ \bigcup_{i=1}^{n-k+1} + (i, O(n-i, k-1)) & \text{if } k > 1 \end{cases}$$

References

1. Donald E. Knuth, *The Art of Computer Programming*, Volume 4, Fascicle 3: Generating All Combinations and Partitions, 2005
2. Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest and Clifford Stein, *Introduction to Algorithms*, 2nd ed., MIT Press, 2001
3. George E. Andrews and Kimmo Eriksson, *Integer Partitions*, Cambridge University Press 2004
4. George E. Andrews, *The Theory of Partitions*, Cambridge University Press 1998