0.1 Generating Partitions

- *Algorithm:* descendingDecomp (algo. 1), ascendingDecomp (algo. 2)
- *Input:* a positive integer n
- *Complexity:* $\mathcal{O}(2^n)$ for both
- *Data structure compatibility:* N/A
- *Common applications:* nuclear fission

Problem. Generating Partitions

Given a positive integer n, generate all partitions where the sum of numbers in this partition is n.

Description

Integer partition of n are sets of non-zero integers that add up to exact n. A popular solution is to generate the partition in lexicographically decreasing order. For example, the integer partition of 6 in decreasing order is $[6], [5, 1], [4, 2], [4, 1, 1], [3, 3], [3, 2, 1], [3, 1, 1, 1], [2, 2, 2], [2, 2, 1, 1], [2, 1, 1, 1, 1], [1, 1, 1, 1, 1, 1]$. However, another solution is to generate in ascending order (the reverse generating order of the above example), which has high efficiency with *Python* unique feature. And the time complexity for both is $\mathcal{O}(2^n)$ because of the execution time mainly determined by the partition function, which would be discussed later.

Descending Order

The partition is generated in lexicographically descending order such that generating all partitions $a_1 \geq a_2 \geq ... \geq a_m \geq 1$ where $a_1 + a_2 + ... + a_m = n$ and $1 \leq m \leq n$. The key is to find next partition based on the current partition, so as to decompose. The first partition is $\{n\}$ itself, and then following the rule that subtract the smallest but larger than one value by 1 and collect all the 1's so to match the new smallest part larger than 1 [**1**]. Assume the current partition is array PC and the smallest but larger than 1 value is at index k, and denote the sum of all the 1's as $sumOne$.The next partition to generate based on the current partition PC has two cases

- If $(PC[k] - 1) \geq (sumOne + 1)$, $P[k] -= 1$ and put $sumOne + 1$ at $PC[k + 1]$, and then the sorted order is realized (e.g. $PC = [5, 1]$, $k = 0, P[k] -= 1, P[k + 1] = sumOne + 1 = 1 + 1$, next partition is $[4, 2]$).
- If $PC[k] < sumOne$, let $PC[k + 1] = PC[k], sumOne - = PC[k]$ and increment k until $(PC[k] - 1) \geq (sumOne + 1)$. The step is similar to divide $sumOne + 1$ by $P[k] - 1$ and put the division result at next $P[k] - 1$ positions (e.g. $PC = [3, 1, 1, 1], k = 0, sumOne + 1 = 4/(PC[k] - 1) = 4/2 = 2$, the next partition is $[2, 2, 2]$).

The partition function $p(n)$ represents the number of possible partitions of a non-negative integer n. However, no exact formula for this function is known. It grows as an exponential function as the square root of n [**2**]. So the time complexity of this algorithm is $\mathcal{O}(2^n)$.

Algorithm 1: DescendingDecomp

Input : An integer n
Output: All integer partitions of n, stored in *result*

```
1  PC ← a size-n array with 0
2  k ← 0
3  PC[k] ← n
4  while True do
5  |  push PC[0 : k + 1] into result
6  |  break the loop if k < 0
7  |  sumOne ← number of 1 in PC[0 : k + 1]
8  |  PC[k] ← PC[k] − 1
9  |  sumOne ← sumOne + 1
10 |  while sumOne > P[k] do
11 |  |  PC[k + 1] ← P[k]
12 |  |  sumOne ← sumOne − P[k]
13 |  |  k ← k + 1
14 |  end while
15 |  PC[k + 1] ← sumOne
16 |  k ← k + 1
17 end while
18 return result
```

Ascending Order

This algorithm will begin with an array of 1 with size n, then add last two elements in the array. When the two numbers add up to 2, there is no other option (as they could only be decomposed into two 1). However, for other added up number, other decompose option need to consider. A recursive algorithm is discussed here. With the smallest part at least m, prepending m to all ascending compositions of $n - m$. As m could not be less than 1 or greater than $\lfloor n/2 \rfloor$, m range from 1 to $\lfloor n/2 \rfloor$. The process is completed by visiting the single composition of n.

Algorithm 2: AscendingDecomp

Input : An integer n
Output: All integer partitions of n, stored in *result*

```
1  a_1 ← 0
2  k ← 2
3  a_2 ← n
4  while k ≠ 1 do
5  |  y ← a_k − 1
6  |  k ← k − 1
7  |  x ← a_k + 1
8  |  while x ≤ y do
9  |  |  a_k ← x
10 |  |  y ← y − x
11 |  |  k ← k + 1
12 |  end while
13 |  a_k ← x + y
14 |  store a_1, ..., a_k into result
15 end while
16 return result
```

This algorithm is constant Amortised time, so the average computation per partition is constant. The algorithm could be further modified to have a better efficiency[**3**]. This extra efficiency is gained through the structure of the neighbor partitions generating in the ascending process. For example, consider the following partition of

$n = 10$.

$$1+1+2+6, 1+1+3+5, 1+1+4+4$$

This transition could be made very efficiently as we only need to add one and subtract one to the second last and last part respectively. Based on this, the algorithm is modified as All three algorithm is implemented

Algorithm 3: AscendingDecomp2

Input : An integer n
Output: All integer partitions of n, stored in *result*

```
1  a_1 ← 0
2  k ← 2
3  y ← n − 1
4  while k ≠ 1 do
5      k ← k − 1
6      x ← a_k + 1
7      while 2x ≤ y do
8          a_k ← x
9          y ← y − x
10         k ← k + 1
11     end while
12     l ← k + 1
13     while x ≤ y do
14         a_k ← x
15         a_l ← y
16         store a_1, ..., a_l into result
17         x ← x + 1
18         y ← y − 1
19     end while
20     y ← y + x − 1
21     a_k ← y + 1
22     store a_1, ..., a_k into result
23 end while
24 return result
```

in Python and the Ascending Decomposition is implemented using generator.The simulation result shows that using Python generator would improve the program efficiency greatly, which meets the assumption.

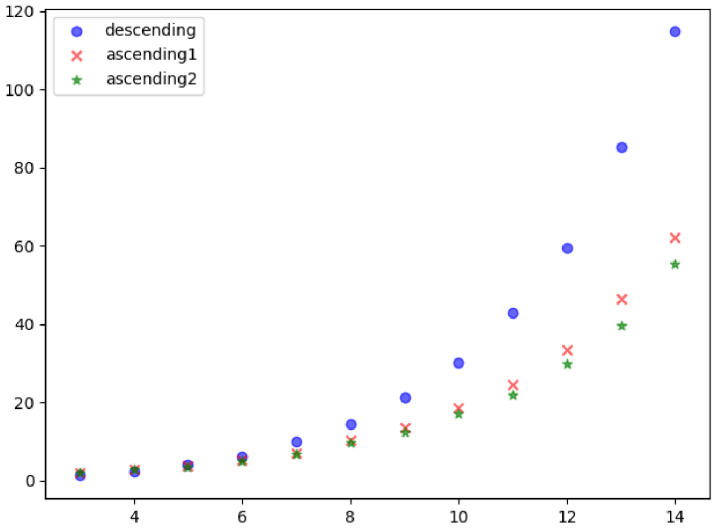

Figure 1: Time of different algorithm