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 [3]. 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 [1]. 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*

```
PC ← a size-n array with 0
k ← 0
PC[k] ← n
while True do
    push PC[0 : k + 1] into result
    break the loop if k < 0
    sumOne ← number of 1 in PC[0 : k + 1]
    PC[k] ← PC[k] − 1
    sumOne ← sumOne + 1
    while sumOne > P[k] do
        PC[k + 1] ← P[k]
        sumOne ← sumOne − P[k]
        k ← k + 1
    end while
    PC[k + 1] ← sumOne
    k ← k + 1
end while
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*

```
a_1 ← 0
k ← 2
a_2 ← n
while k ≠ 1 do
    y ← a_k − 1
    k ← k − 1
    x ← a_k + 1
    while x ≤ y do
        a_k ← x
        y ← y − x
        k ← k + 1
    end while
    a_k ← x + y
    store a_1, ..., a_k into result
end while
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[2]. 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

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
```

All three algorithm is implemented 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. Also, if use the second ascending decomposition algorithm, the algorithm would have sightly better performance.

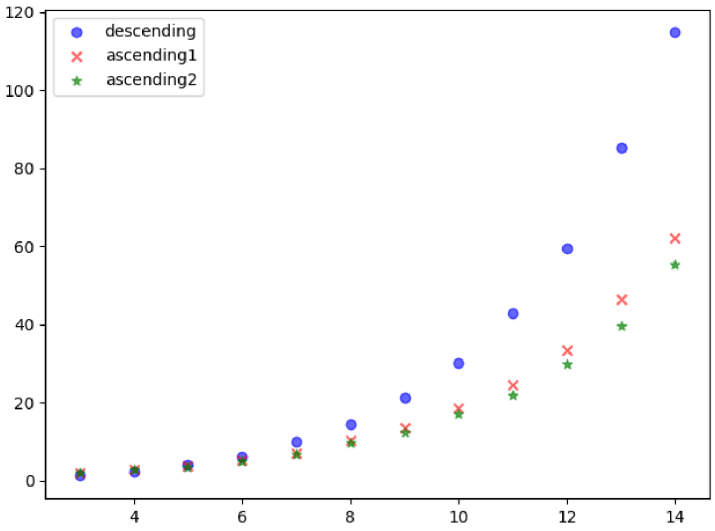

Figure 1: Time of different algorithm

References.

[1] Donald E.Knuth. *The Art of Computer Programming*. Addison-Eesley, 2005 (cit. on p. 1).

[2] Jerome Kelleher. *Generating All Partitions: A Comparison Of Two Encodings*. 2015. URL: https://arxiv.org/abs/0909.2331v2 (cit. on p. 2).

[3] Steven S. Skiena. *The Algorithm Design Manual – Generating Partitions*. 1997. URL: https://www8.cs.umu.se/kurser/TDBAfl/VT06/algorithms/BOOK/BOOK4/NODE152.HTM (cit. on p. 1).

0.2 Generating Subsets

- *Algorithm:* Backtracking (algo. 4), binaryCounting (algo. 5), grayCode (algo. 6)
- *Input:* A set of integers with length n
- *Complexity:* $\mathcal{O}(2^n)$ with Backtracking, $\mathcal{O}(n \cdot 2^n)$ with binaryCounting and grayCode
- *Data structure compatibility:* stack can be used with Backtracking, no specific data structure needed with binaryCounting
- *Common applications:* vertex cover, knapsack, set packing

Problem. Generating Subsets

Given a set of distinct integers, generate all possible subsets.

Description

As specified in the overview, the input is a set of distinct integers. Although *set* is defined as consisting of non-repetitive elements, distinct is emphasized to avoid misunderstanding. Input could also be a positive integer n, which represents the set $\{1, 2, ..., n\}$. And this is could be included in the *a set of distinct integers*.

A more general input would be a set of *any type items*, such as set $\{a, b, c\}$. As the algorithm would be same as for a set of integers, this project will use a set of integers for simpler explanation.

The output is all the subsets of the given set S with length n (power set $P(S)$), so there are 2^n subsets in $P(S)$. For successful generating, the key is to establish a numerical sequence among all subsets and there are three primary alternatives [1]

- *Lexicographic order*: the most intuitive order for generating combinatorial objects, as human will generally use. For example, the eight subsets of $\{1, 2, 3\}$ in lexicographic order would be $\{\}, \{1\}, \{1, 2\}, \{1, 2, 3\}, \{1, 3\}, \{2\}, \{2, 3\}, \{3\}$. As each time we find a previous number "back", the **backtracking** algorithm will find the subset in this order.
- *Binary counting*: Based on the binary number representation of the number of the subset. For example,for set $\{1, 2, 3\}$, the binary number representation from 0 to $2^n - 1 = 7$ would be $000, 001, 010, 011, 100, 101, 110, 111$, which corresponds to $\{\}, \{3\}, \{2\}, \{2, 3\}, \{1\}, \{1, 3\}, \{1, 2\}, \{1, 2, 3\}$. As the subset could be decoded through the number from 0 to $2^n - 1 = 7$, the **binaryCoutning** algorithm would find subsets in this order.
- *Gray code*: A subset sequence as the adjacent subsets differ by insertion or deletion of exactly one item. The gray code for 3-bits is $000, 001, 011, 010, 110, 111, 101, 100$. Compared with *binarycounting*, such sequence could not be obtained directly. With this sequence, generate the subsets based on this sequence will have same procedure as based on *binaryCounting*.

Backtracking

The key idea is that with n elements in the set, for each element, there are two choices: either include or exclude the element in the subset[2].

To analyse time complexity, because not all the sub-problems would have roughly equal size, Master methodis not applicable. The number of execution operations could be represented as

$$T(n) = \sum_{i=1}^{n-1} T(n-i) + c$$

Algorithm 4: Backtracking(*set*)

Input : A set of distinct integers *set*
Output: *result*, store all the subsets of *set*

```
1  Function subsetRec(set, subset, result, index):
2      push subset into result
3      for i ← index to len(set) do
4          push set[i] into subset
5          subsetRec(set, subset, result, index + 1)
6          pop out the last element of subset                    /* exclude set[i] and backtracking */
7      end for
8      return
9  end
10 subsetRec(set, [ ], [[ ]], 0)
11 return result
```

which is caused by the *for* loop that from 0 to *len*(*set*). As each term could be expanded following the same equation such that

$$T(n-1) = T(n-2) + T(n-1) + ... + T(1)$$

Then we could get

$$\begin{aligned} T(n) &= T(n-1) + T(n-2) + ... + T(1) + c \\ &= \big(T(n-2) + T(n-3) + ... + T(1) + c\big) + T(n-2) + ... + T(1) + c \\ &= 2 \times T(n-1) \\ T(n-1) &= 2 \times T(n-2) \\ &... \\ T(2) &= 2T(1) \\ T(1) &= T(0) \end{aligned}$$

Because the equation only differs from constant term, the equation above is acceptable. As we have $T(1) = \mathcal{O}(1)$,

$$T(n) = 2^{n-1}T(1) = \mathcal{O}(2^n).$$

This result could also be obtained by a very intuitive induction. Such that each step there are two choices and n elements (to take both choices) in total, the time complexity should be $\mathcal{O}(2^n)$

Binary Counting

The number of all subsets of a set with size n is 2^n. Binary counter approach could be used, as any unique binary string of length n represent a unique subset. So we could start from 0 and end with $2^n - 1$, and for every bit that set as 1, push the number represent by this bit into subset.

Time complexity for this algorithm is easier to analyse. The outer loop will execute for 2^n times and inner will execute for n times. So the total time complexity is $\mathcal{O}(n \cdot 2^n)$

Algorithm 5: binaryCounting(*set*)

Input : A set of distinct integers *set*
Output: *result*, store all the subsets of *set*

```
1  n ← len(set)
2  for count ← 0 to 2^n − 1 do
3  |  subset ← [ ]
4  |  for bit ← 0 to n do
5  |  |  if count&(1 << bit)                    /* every time left shift 1 by bit */ then
6  |  |  |  push set[bit] into subset                 /* find the bit set as 1 in count */
7  |  |  end if
8  |  end for
9  |  push subset into result
10 end for
11 return result
```

Gray Code

Recursion and iteration both could be used to generate gray code with given length n. As they have similar idea, here only the iteration method is presented.

Algorithm 6: grayCode(*set*)

Input : A set of distinct integers *set*
Output: *result*, store all the subsets of *set*

```
1  Function IterationCode(n):
2  |  arr ← [[ ]]
3  |  push "0","1" into arr[0], arr[1]                          /* one-bit 0 or 1 */
4  |  for i ← 2 to 2^n by ×2 do
5  |  |  for j ← i − 1 to 0 by −1 do
   |  |  |          /* the previous generated code is added again in reverse order */
6  |  |  |  push arr[j] into arr
7  |  |  end for
8  |  |  for j ← 0 to i by +1 do
9  |  |  |  add "0" ahead arr[j]                                 /* 0 to first half */
10 |  |  end for
11 |  |  for j ← i to i ∗ 2 by +1 do
12 |  |  |  add "1" ahead arr[j]                                /* 1 to second half */
13 |  |  end for
14 |  end for
15 |  return arr
16 end
17 code ← IterationCode(len(set))
18 for j ← 0 to 2^n − 1 do
19 |  subset ← [ ]                              /* same procedure as in binaryCounting */
20 |  for bit ← 0 to n do
21 |  |  if code[j]&(1 << bit) then
22 |  |  |  push set[bit] into subset
23 |  |  end if
24 |  end for
25 |  push subset into result
26 end for
27 return result
```

Time complexity for this algorithm is $\mathcal{O}(n \cdot 2^n)$.

For generating the code, the out loop will execute for $(n-1)$ times. For three inner loop, as they both use i and

$2 + 2^2 + ... + 2^{n-1} = \mathcal{O}(2^n)$. So total time complexity for generating code is$\mathcal{O}(n \cdot 2^n)$. Generating the subsets based on the code is $\mathcal{O}(n \cdot 2^n)$. So we could get total time complexity $\mathcal{O}(n \cdot 2^n)$.

References.

[1] Steven S. Skiena. *The Algorithm Design Manual – Generating Subsets*. 1997. URL: https://www8.cs.umu.se/kurser/TDBAfl/VT06/algorithms/BOOK/BOOK4/NODE152.HTM (cit. on p. 5).

[2] Arshpreet Soodan. *Backtracking to find all subsets*. 2020. URL: https://www.geeksforgeeks.org/backtracking-to-find-all-subsets/ (cit. on p. 5).