0.1 Generating Subsets

- *Algorithm:* Backtracking (algo. 1), binaryCounting (algo. 2), grayCode (algo. 3)
- *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 [**1**]

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.

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 1: 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 $\mathcal{O}(n \cdot 2^n)$

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 2: 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
```

Algorithm 3: 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 $\mathcal{O}(m \cdot 2^n)$. So we could get total time complexity $\mathcal{O}(n \cdot 2^n)$.