Group 2 - Programming Assignment #1

Grop Number	Group Members
2	**Hansong Qi, 99901449** **Ruibo Jing, 999015852** **Zhihao Zhao, 999014525** **Zhi Mai, 999009350** **Hongwei Guo, 999014780**

Exercise A (by Hansong Qi and Zhihao Zhao)

Since every subset of a n-set has different characteristic vector, for instance [1, 0, 0] stands for [1], [1, 0, 1] stands for [1, 3], each vector is a binary number of size n, and ranking algorithm is like a conversion from the binary representation to the number.

The pseudo code is:

```
SubsetRank(n, T):
    r ← 0
    for i from 1 to n do
        r ← 2r
        if i in T
            then r ← r + 1
    return r
```

Similarly, unranking algorithm is like a conversion from number to its binary representation, and use it to generate the subset we want.

The pseudo code is:

```
SubsetUnrank(n, r):
    T = ϕ
    for i from n to 1 do
        if r mod 2 = 1
            then T ← T ∪ {i}
        r ← ⟦r/2⟧
    return T
```

For a given subset, the successor algorithm needs to give us its next subset, to do that we can think of increment of a binary number.

The code is:

```
Successor(n, T): // Successor algorithm
    i ← 0
    while (i ≤ n - 1) and (n - i ∈ T) do:
        T ← T \ {n - i}
        i ← i + 1
    if i ≤ n – i
        then T ← T ∪ {n - i}
    return T
```

Since we have the successor algorithm, we can use it to generate a collection of subsets of n-set. We know that a set of cardinality n has 2^n subsets, therefore we need to run the successor algorithm for 2^n times in order to find all subsets.

The pseudo code is:

```
Collection(n): // Generate all subsets of n_set using successor algorithm
    subsets ← ϕ
    subset ← {1,2,...,n}
    for i from 1 to 2^n:
        subset ← Successor(n, subset)
        subsets ← subsets + {subset}
    return subsets
```

Result for n = 4 is:

```
the set is [[], [4], [3], [3, 4], [2], [2, 4], [2, 3], [2, 3, 4], [1], [1, 4], [1, 3], [1, 3, 4], [1, 2],
[1, 2, 4], [1, 2, 3], [1, 2, 3, 4]]
The rank of [] is 0 the unrank of 0 is []
The rank of [4] is 1 the unrank of 1 is [4]
The rank of [3] is 2 the unrank of 2 is [3]
The rank of [3, 4] is 3 the unrank of 3 is [3, 4]
The rank of [2] is 4 the unrank of 4 is [2]
The rank of [2, 4] is 5 the unrank of 5 is [2, 4]
The rank of [2, 3] is 6 the unrank of 6 is [2, 3]
The rank of [2, 3, 4] is 7 the unrank of 7 is [2, 3, 4]
The rank of [1] is 8 the unrank of 8 is [1]
The rank of [1, 4] is 9 the unrank of 9 is [1, 4]
The rank of [1, 3] is 10 the unrank of 10 is [1, 3]
The rank of [1, 3, 4] is 11 the unrank of 11 is [1, 3, 4]
The rank of [1, 2] is 12 the unrank of 12 is [1, 2]
The rank of [1, 2, 4] is 13 the unrank of 13 is [1, 2, 4]
The rank of [1, 2, 3] is 14 the unrank of 14 is [1, 2, 3]
The rank of [1, 2, 3, 4] is 15 the unrank of 15 is [1, 2, 3, 4]
```

Result for n = 5 is:

```
the set is [[], [5], [4], [4, 5], [3], [3, 5], [3, 4], [3, 4, 5], [2], [2, 5], [2, 4], [2, 4, 5], [2, 3],
[2, 3, 5], [2, 3, 4], [2, 3, 4, 5], [1], [1, 5], [1, 4], [1, 4, 5], [1, 3], [1, 3, 5], [1, 3, 4], [1, 3, 4
, 5], [1, 2], [1, 2, 5], [1, 2, 4], [1, 2, 4, 5], [1, 2, 3], [1, 2, 3, 5], [1, 2, 3, 4], [1, 2, 3, 4, 5]]
The rank of [] is 0 the unrank of 0 is []
The rank of [5] is 1 the unrank of 1 is [5]
The rank of [4] is 2 the unrank of 2 is [4]
The rank of [4, 5] is 3 the unrank of 3 is [4, 5]
The rank of [3] is 4 the unrank of 4 is [3]
The rank of [3, 5] is 5 the unrank of 5 is [3, 5]
The rank of [3, 4] is 6 the unrank of 6 is [3, 4]
The rank of [3, 4, 5] is 7 the unrank of 7 is [3, 4, 5]
The rank of [2] is 8 the unrank of 8 is [2]
The rank of [2, 5] is 9 the unrank of 9 is [2, 5]
The rank of [2, 4] is 10 the unrank of 10 is [2, 4]
The rank of [2, 4, 5] is 11 the unrank of 11 is [2, 4, 5]
The rank of [2, 3] is 12 the unrank of 12 is [2, 3]
The rank of [2, 3, 5] is 13 the unrank of 13 is [2, 3, 5]
The rank of [2, 3, 4] is 14 the unrank of 14 is [2, 3, 4]
The rank of [2, 3, 4, 5] is 15 the unrank of 15 is [2, 3, 4, 5]
The rank of [1] is 16 the unrank of 16 is [1]
The rank of [1, 5] is 17 the unrank of 17 is [1, 5]
The rank of [1, 4] is 18 the unrank of 18 is [1, 4]
The rank of [1, 4, 5] is 19 the unrank of 19 is [1, 4, 5]
The rank of [1, 3] is 20 the unrank of 20 is [1, 3]
The rank of [1, 3, 5] is 21 the unrank of 21 is [1, 3, 5]
The rank of [1, 3, 4] is 22 the unrank of 22 is [1, 3, 4]
The rank of [1, 3, 4, 5] is 23 the unrank of 23 is [1, 3, 4, 5]
The rank of [1, 2] is 24 the unrank of 24 is [1, 2]
The rank of [1, 2, 5] is 25 the unrank of 25 is [1, 2, 5]
The rank of [1, 2, 4] is 26 the unrank of 26 is [1, 2, 4]
The rank of [1, 2, 4, 5] is 27 the unrank of 27 is [1, 2, 4, 5]
The rank of [1, 2, 3] is 28 the unrank of 28 is [1, 2, 3]
The rank of [1, 2, 3, 5] is 29 the unrank of 29 is [1, 2, 3, 5]
The rank of [1, 2, 3, 4] is 30 the unrank of 30 is [1, 2, 3, 4]
The rank of [1, 2, 3, 4, 5] is 31 the unrank of 31 is [1, 2, 3, 4, 5]
```

Result for n = 6 is:

```
the set is [[], [6], [5], [5, 6], [4], [4, 6], [4, 5], [4, 5, 6], [3], [3, 6], [3, 5], [3, 5, 6], [3, 4],
[3, 4, 6], [3, 4, 5], [3, 4, 5, 6], [2], [2, 6], [2, 5], [2, 5, 6], [2, 4], [2, 4, 6], [2, 4, 5], [2, 4, 5
, 6], [2, 3], [2, 3, 6], [2, 3, 5], [2, 3, 5, 6], [2, 3, 4], [2, 3, 4, 6], [2, 3, 4, 5], [2, 3, 4, 5, 6],
[1], [1, 6], [1, 5], [1, 5, 6], [1, 4], [1, 4, 6], [1, 4, 5], [1, 4, 5, 6], [1, 3], [1, 3, 6], [1, 3, 5],
[1, 3, 5, 6], [1, 3, 4], [1, 3, 4, 6], [1, 3, 4, 5], [1, 3, 4, 5, 6], [1, 2], [1, 2, 6], [1, 2, 5], [1, 2,
 5, 6], [1, 2, 4], [1, 2, 4, 6], [1, 2, 4, 5], [1, 2, 4, 5, 6], [1, 2, 3], [1, 2, 3, 6], [1, 2, 3, 5], [1,
 2, 3, 5, 6], [1, 2, 3, 4], [1, 2, 3, 4, 6], [1, 2, 3, 4, 5], [1, 2, 3, 4, 5, 6]]
The rank of [] is 0 the unrank of 0 is []
The rank of [6] is 1 the unrank of 1 is [6]
The rank of [5] is 2 the unrank of 2 is [5]
The rank of [5, 6] is 3 the unrank of 3 is [5, 6]
The rank of [4] is 4 the unrank of 4 is [4]
The rank of [4, 6] is 5 the unrank of 5 is [4, 6]
The rank of [4, 5] is 6 the unrank of 6 is [4, 5]
The rank of [4, 5, 6] is 7 the unrank of 7 is [4, 5, 6]
The rank of [3] is 8 the unrank of 8 is [3]
The rank of [3, 6] is 9 the unrank of 9 is [3, 6]
The rank of [3, 5] is 10 the unrank of 10 is [3, 5]
The rank of [3, 5, 6] is 11 the unrank of 11 is [3, 5, 6]
The rank of [3, 4] is 12 the unrank of 12 is [3, 4]
The rank of [3, 4, 6] is 13 the unrank of 13 is [3, 4, 6]
The rank of [3, 4, 5] is 14 the unrank of 14 is [3, 4, 5]
The rank of [3, 4, 5, 6] is 15 the unrank of 15 is [3, 4, 5, 6]
The rank of [2] is 16 the unrank of 16 is [2]
The rank of [2, 6] is 17 the unrank of 17 is [2, 6]
The rank of [2, 5] is 18 the unrank of 18 is [2, 5]
The rank of [2, 5, 6] is 19 the unrank of 19 is [2, 5, 6]
The rank of [2, 4] is 20 the unrank of 20 is [2, 4]
The rank of [2, 4, 6] is 21 the unrank of 21 is [2, 4, 6]
The rank of [2, 4, 5] is 22 the unrank of 22 is [2, 4, 5]
The rank of [2, 4, 5, 6] is 23 the unrank of 23 is [2, 4, 5, 6]
The rank of [2, 3] is 24 the unrank of 24 is [2, 3]
The rank of [2, 3, 6] is 25 the unrank of 25 is [2, 3, 6]
The rank of [2, 3, 5] is 26 the unrank of 26 is [2, 3, 5]
The rank of [2, 3, 5, 6] is 27 the unrank of 27 is [2, 3, 5, 6]
The rank of [2, 3, 4] is 28 the unrank of 28 is [2, 3, 4]
The rank of [2, 3, 4, 6] is 29 the unrank of 29 is [2, 3, 4, 6]
The rank of [2, 3, 4, 5] is 30 the unrank of 30 is [2, 3, 4, 5]
The rank of [2, 3, 4, 5, 6] is 31 the unrank of 31 is [2, 3, 4, 5, 6]
```

```
The rank of [1] is 32 the unrank of 32 is [1]
The rank of [1, 6] is 33 the unrank of 33 is [1, 6]
The rank of [1, 5] is 34 the unrank of 34 is [1, 5]
The rank of [1, 5, 6] is 35 the unrank of 35 is [1, 5, 6]
The rank of [1, 4] is 36 the unrank of 36 is [1, 4]
The rank of [1, 4, 6] is 37 the unrank of 37 is [1, 4, 6]
The rank of [1, 4, 5] is 38 the unrank of 38 is [1, 4, 5]
The rank of [1, 4, 5, 6] is 39 the unrank of 39 is [1, 4, 5, 6]
The rank of [1, 3] is 40 the unrank of 40 is [1, 3]
The rank of [1, 3, 6] is 41 the unrank of 41 is [1, 3, 6]
The rank of [1, 3, 5] is 42 the unrank of 42 is [1, 3, 5]
The rank of [1, 3, 5, 6] is 43 the unrank of 43 is [1, 3, 5, 6]
The rank of [1, 3, 4] is 44 the unrank of 44 is [1, 3, 4]
The rank of [1, 3, 4, 6] is 45 the unrank of 45 is [1, 3, 4, 6]
The rank of [1, 3, 4, 5] is 46 the unrank of 46 is [1, 3, 4, 5]
The rank of [1, 3, 4, 5, 6] is 47 the unrank of 47 is [1, 3, 4, 5, 6]
The rank of [1, 2] is 48 the unrank of 48 is [1, 2]
The rank of [1, 2, 6] is 49 the unrank of 49 is [1, 2, 6]
The rank of [1, 2, 5] is 50 the unrank of 50 is [1, 2, 5]
The rank of [1, 2, 5, 6] is 51 the unrank of 51 is [1, 2, 5, 6]
The rank of [1, 2, 4] is 52 the unrank of 52 is [1, 2, 4]
The rank of [1, 2, 4, 6] is 53 the unrank of 53 is [1, 2, 4, 6]
The rank of [1, 2, 4, 5] is 54 the unrank of 54 is [1, 2, 4, 5]
The rank of [1, 2, 4, 5, 6] is 55 the unrank of 55 is [1, 2, 4, 5, 6]
The rank of [1, 2, 3] is 56 the unrank of 56 is [1, 2, 3]
The rank of [1, 2, 3, 6] is 57 the unrank of 57 is [1, 2, 3, 6]
The rank of [1, 2, 3, 5] is 58 the unrank of 58 is [1, 2, 3, 5]
The rank of [1, 2, 3, 5, 6] is 59 the unrank of 59 is [1, 2, 3, 5, 6]
The rank of [1, 2, 3, 4] is 60 the unrank of 60 is [1, 2, 3, 4]
The rank of [1, 2, 3, 4, 6] is 61 the unrank of 61 is [1, 2, 3, 4, 6]
The rank of [1, 2, 3, 4, 5] is 62 the unrank of 62 is [1, 2, 3, 4, 5]
The rank of [1, 2, 3, 4, 5, 6] is 63_the unrank of 63 is [1, 2, 3, 4, 5, 6]
```

Exercise B (by Zhi Mai)

(1) Notation

Suppose $\sigma_1 = [t_1, t_2, \ldots, t_k], \sigma_2 = [r_1, r_2, \ldots, r_k]$ are permutations of k-set. $\sigma_1 > \sigma_2$ if $t_i > r_i$,

$$i = \min\{1 \leqslant i \leqslant k \mid r_i \neq t_i\}. \tag{1}$$

If the elements of σ1 and σ2 are the same, we compare their colors.

Since the number of colors is 2, we can define *color1* to be 1, *color2* to be 0. Then the color of each element in a permutation can be mapped to $\{0, 1\}$. So, the whole color of a permutation can be represented as a **binary string**.

For example, the color of 1432 is 1011 where *blue* represents 0 and *red* represents 1. In other words, an element of 2-color-k-permutation is

$$\{[a_1, \ldots, a_k], b_1 b_2 \ldots b_k\},\ b_i \in \{0, 1\}. \tag{2}$$

Then if we compare the color of σ_1 and σ_2, we just compare the binary number of them and see which is bigger.

(2) Problem Analysis

i.

For a rank, since there are 2^k colors for each permutation, we need take the remainder r and the quotient q of $\frac{\text{rank}}{2^k}$. The number q represents the **rank of permutation**, and r represents **the rank of color**.

To find the permutation corresponds to r, we can fix elements from left to right. Notice that each element of k-set appears only once, if the $i-th$ element of a permutation is fixed, the number of possible remaining permutations is $(k–i)!$.

When we fix the first element t_1, we take the quotient r_1 of r $(k–1)!$ and round r_1 up (since $ri \gg 1$). Then we look for the second element t_2, which is in range

$$(t1 \cdot (k-1)! + 1,\ (t1+1) \cdot (k-1)! - 1). \tag{3}$$

So, to fix t_2, we need to take remainder r_2 of $\frac{r_1}{(k-1)!}$. Also, the $i-th$ element can't coincide with previous all element. So, for 2nd element, if it coincides with previous all elements, we need to add it with 1 until it's a new element. Keep this process until processing all elements. In particular, if $r_i = 0$, it means the remaining part is the biggest permutation of current remaining elements. In other words, the remaining part is a decreasing sequence.

Finally, turn the rank of color into binary number.

The pseudo code is as follows:

```
Class cp: {
    numberOfColor = 2;
    size = k;
    int permutation[size];
    int color[size]
}
```

```
Unrank(n):
    Rank_permutation <- n / 2^k (round up); Rank_color <- n % 2^k;
    cp.color = encoder(cp, 2^(Rank_color) - 1);
    list = [1,..,k]
    for i <- 1 to k - 1 do:
        if (Rank_permutation == 0):
            for j <- i to k do:
                permutation[j] = list[len(list) - j - i];
            break;
        position = Rank_permutation / (k - i)! (round up);
        cp.permutation[i] = list[position];
        delete list[position];
        Rank_permutation = Rank_permutation % (k - i)!;
    cp.permutation[k] = list[1];
    return cp;
```

```
encoder(cp, n):
    remainder <- n;
    for i <- k to 1 do:
        cp.color[i] = remainder % 2;
        remainder = remainder \ 2 (round down);
    return cp.color;
```

ii.

To find the rank of a color-permutation(cp), we need to find the ranks of color and permutation and add 1, since

$$\text{rank} = \#\{\text{color-permutation precede cp}\} + 1. \tag{4}$$

For $i-th$ element, we need to find all natural numbers smaller than ith element. As previous saying, $i-th$ element can't coincide with previous all elements and the number of possible permutations is $(k-i)!$. So, the number of possible permutations corresponding to $i-th$ element is

$$\#\{m | m \text{ not coincides with previous all element and } m < \text{i-th element}\} \cdot (k-i). \tag{5}$$

And we add the numbers from 1 to k, that is,

$$\sum_{i=1}^{k} \#\{m | m \text{ not coincides with previous all element and } m < \text{i-th element}\} \cdot (k–i)!. \tag{6}$$

And we turn the binary string of color into integer. Since for each permutation there are 2^k colors, the rank is

$$2^k \cdot \sum_{i=1}^{k} \#\{m | m \text{ not coincides with previous all element and } m < \text{i-th element}\} \cdot (k–i)! + \text{decoder}(\text{cp.color}) + 1. \tag{7}$$

The pseudo code is as follows:

```
Rank(cp):
    Rank1 = 0;
    Set = {};
    for i <- 1 to k do:
        for j <- I to cp.permutation[i] do:
            if (j not in set)
                Rank1 += (k – i)!;
        Set.add(i);
    Rank2 = decoder(cp);
    return Rank1 * 2^k + Rank2 + 1;
```

```
decoder(cp):
    Rank = 0;
        for i <- 1 to k do:
            if (cp.color[i] == 1):
                Rank += 2^(i – 1);
    return Rank;
```

iii.

To find the successor of a color-permutation(cp), we can first iterate color then iterate permutation. The method of finding successor of permutation is same as in the course. To find successor of binary number, we just invert all the bits from right to left until reaching a bit which is 0. Moreover, the range of binary number is $[0, 2^k–1]$. So, if the binary string is $11…1$, its successor is $00…0$ and we need to find successor of permutation. Otherwise, we just find the successor of binary string.

The psuedo code is as follows:

```
Successor(cp):
    Carry = 0;
    if (cp.color[k] == 0):
        cp.color[k] = 1;
    else:
        for i <- 1 to k do:
            if (cp.color[i] == 1)
                cp.color[i] = 0;
                if (i == k):
                    carry = 1;
            else:
                cp.color[i] = 1;
                break;
    if (carry == 1):
        l = 0;
        for i <- k to 1 do:
```

```
            if (cp.permutation[i - 1] < cp.permutation[i]):
                l = i;
                x = k;
                while(cp.permutation[i - 1] > cp.permutation[x] and x >= 0):
                    x -= 1;
                    if (x == 0):
                        break;
                swap(cp.permutation[i - 1], cp.permutation[x]);
        for j <- l to l + (k - l) \ 2 (round up) do:
            swap(cp.permutation[j], cp.permutation[k - (j - l)]);
    return cp;
```

iv).

So, for a k-permutation, it has 2^k different possibilities of colors. And for a k-set, it has k! permutations. Then for a k-set it has $2^k \cdot k!$ many 2-color-permutations. So, we just need to use successor function $2^k \cdot k!$ times, start from $\{[1, 2, \ldots, k],\ 00\ldots0\}$.

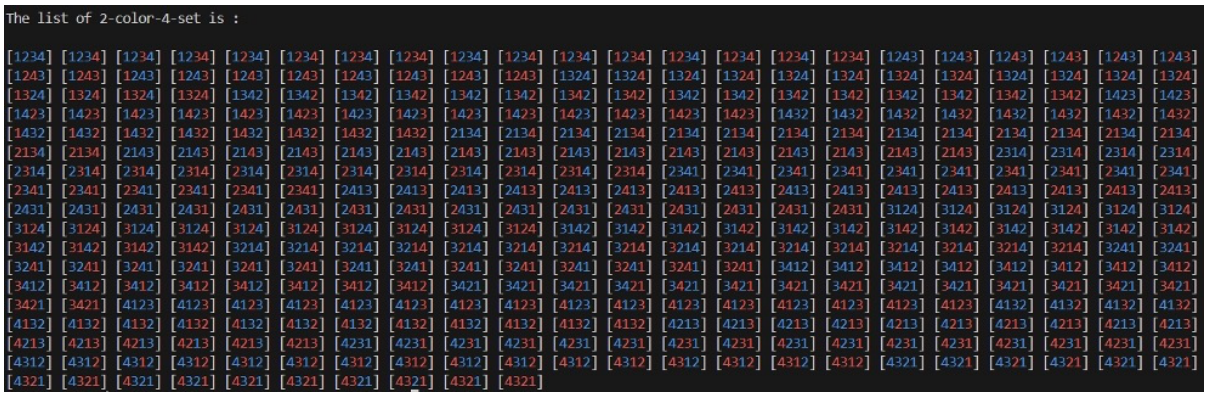
The pseudo code is as follows:

```
List(n):
    cp.size = n;
    cp.permutation = int[n];
    cp.color = int[n];
    for i <- 1 to k do:
        cp.permutation[i] = i;
        cp.color[i] = 0;
    for i <- 1 to 2^k * k! do:
        print(cp);
        successor(cp);
```

(3) Output

```
The rank of [2134]  is :108

The rank of [25341]  is :1462

The rank of [612354]  is :38502

The unrank is : [1342]

The unrank is : [12354]

The unrank is : [123456]
```

```
The sucessor of [3124]  is
[3124]

The sucessor of [35142]  is
[35142]

The sucessor of [654312]  is
[654312]
```



Exercise C (done by Hongwei GUO and Ruibo JING)

Given a list of keys L, we have to generate all possible BSTs from L. We are actually asked to insert all the keys into BST with all possible order of L.

(1) General idea of backtracking

The pseudo code of generating all possible BST by backtracking given a list of keys is as follow:

```
Global: L, N = len(L), BST

// Algorithm 1.
GENERATE_BST(L, CurrKey, Visited, l):

    BST.insert(CurrKey)

    // 1. Check the feasiblity of the current BST.

    if l = N: // all keys in L have been inserted in BST
        BST.print

    // 2. Construct the choice list for the current key.

    Cl = L \ Visited ∪ {CurrKey} // the raletive positions of keys in L should be preserved

    // 3. Assign every key in Cl to CurrKey and call the function recursively.

    for nextKey in Cl:
        GENERATE_BST(L, nextKey, Visited ∪ {CurrKey}, l+1)

    BST.remove(CurrKey)
```

The algorithme works, beceause it actually insert all the keys of L, recursively, into BST with all possible order. The outcome of `Algorithm 1` is listed in `output1.txt`, `output2.txt` and `output3.txt`.

(2) Do Pruning

But we came across a situation that, **it can generate exactlly the same BST even if the keys are inserted in different order.**

That means, we need to do some pruning to the `Algorithm 1` above.

We say two *permutations* of L are **equivalent** iff they generates the same binary tree, keys inserted in BST following the order of keys in them.

The basic idea is, we check whether the BST, which is about to be printed, generated by the current input list (i.e. `Visited U {CurrKey}`, which is actually a perumution of L) has been printed before. If no, then print it and record all input lists equivalent; else, return.

The correspoing pseudo code is as follows:

```
Global: L, N = len(L), BST, Printed_List

// Algorithm 2.
GENERATE_BST_PRUNING(L, CurrKey, Visited, l):

    BST.insert(CurrKey)

    // 1. Check the feasiblity of the current BST.

    if l = N and Visited ∪ {CurrKey} ∉ Printed_List: // check whether BST has been printed
        BST.print
        Equiv_List ← GET_EQUIV_PERMU(BST)
        Printed_List ← Printed_List ∪ Equiv_List

    // 2. Construct the choice list for the current key.

    Cl = L \ Visited ∪ {CurrKey} // the raletive positions of keys in L should be preserved

    // 3. Assign every key in Cl to CurrKey and call the function recursively.

    for nextKey in Cl:
        GENERATE_BST_PRUNING(L, nextKey, Visited ∪ {CurrKey}, l+1)

    BST.remove(CurrKey)
```

The result `Algorithm 2` is listed in `output1_pruning.txt`, `output2_pruning.txt` and `output3_pruning.txt`.

(3) Code

It is very hard to check only by looking at the ordered list of the BST whether it should by prunned or not. We have did a lot of work on that but we failed, so this part we don't demonstrate it here.

Therefore, at every try, we use the global_printed_list to store every things we have printed, so we can ensure no duplicated graph. That is, we prun every duplicated graph.
We use the backtracking algorithm to achieve these requirements.

1. At first step, we insert the CurrKey. If the length is OK and the List does not generate a graph which is already in the global_printed_bst, then we print the list so that we can do checks. After that, we need to meet the requirement of never adding the new list with the same graph. To do so, every time we add a list, we add all its equivalence lists to global_print_bst as well. Then, the next time we deal with a new graph, if it is in the global_print_bst, it means it is equivalent to some previous BST tree in graph, so we prun it. Then, add the count by 1.

2. Construct the choice list for the Currkey. removing everything in the VisitedList and in the Currkey.(Code of this part)
   We initialize the choice list by the original list, removing everything in the Currkey and remove from the Visitedl=List. (Backtracking step)
3. Call the algorithm recursively and assign every key in choice list to CurrKey.
   From the choice list, we obtain the nextkey,and use it to get started. Record the currkey to the VisitedList, add the length by 1 so that in next iteration we can check its length.

In these way, we are able to generate all the BST trees without duplication starting from one specific node.
All we need to do next is to try all the possiblity of initialization of its nodes.