Hanoi University of Science and Technology
School of Information and Communications Technology

# Discrete Mathematics

**Nguyễn Khánh Phương**

**Department of Computer Science**
**School of Information and Communication Technology**
**E-mail: phuongnk@soict.hust.edu.vn**

PART 1

**COMBINATORIAL THEORY**

(Lý thuyết tổ hợp)

PART 2

**GRAPH THEORY**

(Lý thuyết đồ thị)

## Contents of Part 1

Chapter 0: Sets, Relations

Chapter 1: Counting problem

Chapter 2: Existence problem

**Chapter 3: Enumeration problem**

Chapter 4: Combinatorial optimization problem

## Contents of Part 1: Combinatorial Theory

**Chapter 1. Counting problem**

- This is the problem aiming to answer the question: "How many ways are there that satisfy given conditions?" The counting method is usually based on some basic principles and some results to count simple configurations.
- Counting problems are effectively applied to evaluation tasks such as calculating the probability of an event, calculating the complexity of an algorithm

**Chapter 2. Existence problem**

In the counting problem, configuration existence is obvious; in the existence problem, we need to answer the question: "Is there a combinatorial configuration that satisfies given properties ?"

**Chapter 3. Enumeration problem**

This problem is interested in giving all the configurations that satisfy given conditions.

Hanoi University of Science and Technology
School of Information and Communications Technology

Chapter 3

# ENUMERATION PROBLEM

## CONTENTS

1. **Introduction to problem**
2. Generating algorithm
3. Backtracking algorithm

## Introduction to problem

- A problem asking to give a list of all combinations that satisfy a given number of properties is called the combinatorial enumeration problem.
- As the number of combinations to enumerate is usually very large even when the configuration size is not large:
  - The number of permutations of $n$ elements is $n!$
  - The number of subset consisting $m$ elements taken from $n$ elements is $n!/(m!(n-m)!)$

Therefore, it is necessary to have the concept of solving combinatorial-enumeration problems

## Introduction to problem

- The combinatorial enumeration problem is solvable if we can define ***an algorithm*** so that all configurations could be built in turn.
- An enumeration algorithm must satisfy 2 basic requirements:
  - Do not repeat a configuration,
  - Do not miss a configuration.

## CONTENTS

1. Introduction to problem
2. Generating algorithm
3. Backtracking algorithm

## 2. Generating algorithm

**2.1. Algorithm diagram**

2.2. Generate basic combinatorial configurations

- Generate binary strings of length *n*
- Generate m-element subsets of the set of *n* elements
- Generate permutations of *n* elements

## 2.1. Algorithm diagram

The generating algorithm could be applied to solve the combinatorial enumeration problem if the following two conditions are fulfilled:

1) An order can be specified on the set of combinations to enumerate. From there, the first and last configuration could be defined in the order specified.
2) Build the algorithm to give the next configuration of the current configuration (not the final one yet).

The algorithm referred to in condition 2) is called Successive Generation Algorithm

## Generate algorithm

```
void Generate ( )
{
  <Build the first configuration>;
  stop=false;
  while (!stop)
  {
      <Print out the current configuration>;
      if (the current configuration is not the final yet)
         Successive_Generation ( );
      else stop = true;
  }
}
```

<Successive_Generation> is the procedure create the next configuration of the current one in the order specified.

Note: Since the set of combinatorial configurations to enumerate is finite, we can always define an order on them. However, the order should be determined so that the successive generation algorithm could be built.

## 2. Generating algorithm

2.1. Algorithm diagram

2.2. Generate basic combinatorial configurations

– **Generate binary strings of length *n***

– Generate *m*-element subsets of the set of *n* elements

– Generate permutations of *n* elements

NGUYỄN KHÁNH PHƯƠNG
CS-SOICT-HUST

## Generate binary strings of length *n*

**Problem:** Enumerate all binary strings of length $n$:

$$b_1\ b_2 \ldots b_n,\ \text{where } b_i \in \{0, 1\}.$$

Solution:

- Dictionary order:

Consider each binary string $b = b_1\ b_2 \ldots b_n$ as the binary representation of an integer number $p(b)$

We say binary string $b = b_1\ b_2 \ldots b_n$ is ***previous*** binary string $b' = b'_1\ b'_2 \ldots b'_n$ in dictionary order and denote as $b \prec b'$ if $p(b) < p(b')$.

NGUYỄN KHÁNH PHƯƠNG
CS-SOICT-HUST

## Generate binary strings of length *n*

**Example:** When $n$=3, all binary strings of length 3 are enumerated in the dictionary order in the table as following:

| *b* | *p*(*b*) |
|---|---|
| 000 | 0 |
| 001 | 1 |
| 010 | 2 |
| 011 | 3 |
| 100 | 4 |
| 101 | 5 |
| 110 | 6 |
| 111 | 7 |

NGUYỄN KHÁNH PHƯƠNG
CS-SOICT-HUST

## Generate binary strings of length *n*

Successive_Generation algorithm:

- The first string is 0 0 ... 0,
- The last string is 1 1 ... 1.
- Assume $b_1\ b_2 \ldots b_n$ *be the current string:*
  – If this string consists of all 1 → finish,
  – Otherwise, the next string is obtained by adding 1 (module 2, memorize) to the current string.
- Thus, we have following rule to generate the next string:
  – Find the first $i$ (in the order $i = n, n\text{-}1, \ldots, 1$) such that $b_i = 0$.
  – Reassign $b_i = 1$ and $b_j = 0$ for all $j > i$. The newly obtained sequence will be the string to find.

## Generate binary strings of length *n*

- Find the first $i$ (in the order $i = n, n-1, ..., 1$) such that $b_i = 0$.
- Reassign $b_i = 1$ and $b_j = 0$ for all $j > i$. The newly obtained sequence will be the string to find.

Example: consider binary string of length 10: $b =$ 1101011111.
Find the next string:.

- We have $i = 5$. Thus, set:
  - $b_5 = 1$,
  - and $b_i = 0$, $i = 6, 7, 8, 9, 10$

→ We obtain the next string is 1101100000.

```
  1101011111
+          1
------------
  1101100000
```

## Successive_Generation algorithm

```
def  Next_Bit_String( ):
#Generate the next binary string in the dictionary order
#of the current string  b_1 b_2 ... b_n ≠ 1 1 ... 1

    i = n
    while  (b[i] == 1):
        b[i] = 0
        i = i-1
    b[i] = 1
```

- Find the first $i$ (in the order $i = n, n-1, ..., 1$) such that $b_i = 0$.
- Reassign $b_i = 1$ and $b_j = 0$ for all $j > i$. The newly obtained sequence will be the string to find.

## 2. Generating algorithm

2.1. Algorithm diagram

2.2. Generate basic combinatorial configurations

- Generate binary strings of length *n*
- **Generate *m*-element subsets of the set of *n* elements**
- Generate permutations of *n* elements

## Generate *m*-element subsets of set with *n* elements

Problem: Let $X = \{1, 2, ... , n\}$. Enumerate all $m$-element subsets of $X$.

Solution:

- Lexicographic order:

Each $m$-element subset of $X$ could be represented by tuples of $m$ elements

$$a = (a_1, a_2, ... , a_m)$$

satisfying

$$1 \leq a_1 < a_2 < ... < a_m \leq n.$$

## Generate *m*-element subsets of set with *n* elements

We say subset $a = (a_1, a_2,..., a_m)$ is previous subset $a' = (a'_1, a'_2, ... , a'_m)$ in dictionary order and denote as $a \prec a'$, if one could find the index $k$ $(1 \le k \le m)$ such that:

$$a_1 = a'_1 , a_2 = a'_2, . . . , a_{k-1} = a'_{k-1},$$

$$a_k < a'_k .$$

NGUYỄN KHÁNH PHƯƠNG
CS-SOICT-HUST

## Generate *m*-element subsets of set with *n* elements

Example: Enumerate all 3-element subset of $X = \{1, 2, 3, 4, 5\}$ in dictionary order

| | | |
|---|---|---|
| 1 | 2 | 3 |
| 1 | 2 | 4 |
| 1 | 2 | 5 |
| 1 | 3 | 4 |
| 1 | 3 | 5 |
| 1 | 4 | 5 |
| 2 | 3 | 4 |
| 2 | 3 | 5 |
| 2 | 4 | 5 |
| 3 | 4 | 5 |

NGUYỄN KHÁNH PHƯƠNG
CS-SOICT-HUST

## Generate *m*-element subsets of set with *n* elements

Successive_Generation algorithm:

- The first subset is $(1, \quad 2, \quad ... , m)$
- The last subset is $(n-m+1, n-m+2, ..., n)$.
- Assume $a=(a_1, a_2, ... , a_m)$ is the current subset but not the final yet, then its next subset in the dictionary order could be built by using the following rules:
  - Scan from the right to the left of sequence $a_1, a_2,..., a_m$ : find the first element $a_i \neq n-m+i$
  - Replace $a_i$ by $a_i + 1$
  - Replace $a_j$ by $a_i + j - i$, where $j = i+1, i+2,..., m$

NGUYỄN KHÁNH PHƯƠNG
CS-SOICT-HUST

## Generate *m*-element subsets of set with *n* elements

**Example: $n = 6, m = 4$**

Assume the current subset (1, 2, 5, 6), we need to build its next subset in the dictionary order:

- Scan from the right to the left of sequence $a_1, a_2,..., a_m$ : find the first element $a_i \neq n-m+i$
- Replace $a_i$ by $a_i + 1$
- Replace $a_j$ by $a_i + j - i$, where $j = i+1, i+2,..., m$

• We have $i$=2:

Sequence: (1, **2,** 5, 6)

Value $n-m+i$: (3, **4**, 5, 6)

replace $a_2 = a_2+1 = 3$

$a_3 = a_i + j - i = a_2 + 3 - 2 = 4$

$a_4 = a_i + j - i = a_2 + 4 - 2 = 5$

We then obtain its next subset (1, 3, 4, 5).

## Successive_Generation algorithm

```
def Next_Combination( ):
#Generate the next subset in dictionary order of
  #the subset (a_1, a_2,..., a_m)  ≠  (n-m+1,...,n)
   i = m
   while (a[i] == n-m+i):
       i = i-1
   a[i]= a[i] + 1
   for j in range (i+1, m+1):
       a[j] = a[i] + j - i
```

– Find from the right of sequence $a_1, a_2,..., a_m$ the first element $a_i \neq n\text{-}m+i$
– Replace $a_i$ by $a_i + 1$
– Replace $a_j$ by $a_i + j - i$, where $j = i+1, i+2,..., m$

## 2. Generating algorithm

2.1. Algorithm diagram

2.2. Generate basic combinatorial configurations

– Generate binary strings of length $n$

– Generate $m$-element subsets of the set of $n$ elements

– **Generate permutations of $n$ elements**

NGUYỄN KHÁNH PHƯƠNG
CS-SOICT-HUST

## Generate permutations of $n$ elements

**Problem**: Give $X = \{1, 2, ... , n\}$, enumerate all permutations of $n$ elements of $X$.

Solution:

- Dictionary order:
  – Each permutation of $n$ elements of $X$ could be represented by an ordered set of $n$ elements:

$$a = (a_1, a_2, ... , a_n)$$

satisfy

$$a_i \in X, \ \ i = 1, 2,..., n \ , \ a_p \neq a_q, \ p \neq q.$$

NGUYỄN KHÁNH PHƯƠNG
CS-SOICT-HUST

## Generate permutations of $n$ elements

We say permutation $a = (a_1, a_2,..., a_n)$ is previous permutation $a' = (a'_1, a'_2, ... , a'_n)$ in dictionary order and denote as $a \prec a'$, if we could find the index $k$ ($1 \leq k \leq n$) such that:

$$a_1 = a'_1 \, , \, a_2 = a'_2, \, ... \, , \, a_{k-1} = a'_{k-1},$$

$$a_k < a'_k \, .$$

NGUYỄN KHÁNH PHƯƠNG
CS-SOICT-HUST

## Generate permutations of *n* elements

- Example: All permutation of 3 elements of X ={1, 2, 3} could by enumerated in dictionary order as following:

|   |   |   |
|---|---|---|
| 1 | 2 | 3 |
| 1 | 3 | 2 |
| 2 | 1 | 3 |
| 2 | 3 | 1 |
| 3 | 1 | 2 |
| 3 | 2 | 1 |

## Generate permutations of *n* elements

**Successive_Generation algorithm:**

- The first permutation: $(1, 2, \dots, n)$
- The last permutation: $(n, n\text{-}1, \dots, 1)$.
- Assume $a = (a_1, a_2, \dots, a_n)$ is not the last permutation, then its next permutation could be built using the following rules:
  - Find from the right to left: first index $j$ satisfying $a_j < a_{j+1}$ (in other word: $j$ is the max index satisfying $a_j < a_{j+1}$);
  - Find $a_k$ be the smallest number among those on the right of $a_j$ in the sequence and *greater than* $a_j$ ;
  - Swap $a_j$ and $a_k$ ;
  - Invert the sequence from $a_{j+1}$ to $a_n$ .

## Generate permutations of *n* elements

Example: Assume the current permutation (3, 6, 2, 5, 4, 1), need to create its next permutation in the dictionary order.

- Find from the right to left: first index $j$ satisfying $a_j < a_{j+1}$ (in other word: $j$ is the max index satisfying $a_j < a_{j+1}$);
- Find $a_k$ be the smallest number among those on the right of $a_j$ in the sequence and *greater than* $a_j$ ;
- Swap $a_j$ and $a_k$ ;
- Invert the sequence from $a_{j+1}$ to $a_n$ .

- We have index $j = 3$ ($a_3 = 2 < a_4 = 5$).
- The smallest number among those on the right of $a_3$ and greater than $a_3$ is $a_5 = 4$. *Swap* $a_3$ and $a_5$ in the current permutation (3, 6, **2**, 5, **4**, 1)

  we obtain (3, 6, **4**, 5, **2**, 1),
- Finally, invert the sequence $a_4\ a_5\ a_6$ , we obtain the next permutation is (3, 6, 4, **1, 2, 5**).

## Successive_Generation algorithm

```
void Next_Permutation ( )
{
    /*Generate next permutation  $(a_1, a_2, \dots, a_n) \neq (n, n\text{-}1, \dots, 1)$  */
    // Find j being max index satisfying $a_j < a_{j+1}$ :
    j=n-1;  while ($a_j > a_{j+1}$) j=j-1;
   // Find $a_k$ being smallest number among those on the right $a_j$ and greater
    //than $a_j$ :
    k=n;  while ($a_j > a_k$) k=k-1;
    Swap($a_j$, $a_k$); //swap $a_j$ and $a_k$
    // Invert the sequence $a_{j+1}$ to $a_n$ :
    right=n; left=j+1;
    while ( right > left )
    {
         Swap($a_{right}$, $a_{left}$);  // swap $a_{right}$ and $a_{left}$
         right=right-1; left= left+1;
    }
}
```

## Successive_Generation algorithm

```
def Next_Permutation ( ):
#Generate next permutation  $(a_1, a_2, ..., a_n) \neq (n, n-1, ..., 1)$
#Step 1: Find j being max index satisfying $a_j < a_{j+1}$ :
    j = n-1
    while (a[j] > a[j+1]):
        j = j-1
#Step 2: Find $a_k$ being smallest number among those on the right $a_j$ and greater than $a_j$:
    k = n
    while (a[j] > a[k]):
        k = k-1
    Swap(a[j], a[k])

#Step 3: Invert the sequence $a_{j+1}$ to $a_n$ :
    r = n
    l = j+1
    while ( r > l ):
        Swap(a[r], a[l])
        r = r-1
        l = l+1
```

## Exercise

**a)** Let $X = \{1, 4, 5, 6, 7, 8\}$ be the subset of 6 elements of the set $N_9 = \{1, 2, ..., 9\}$. Give 3 next subsets of X in dictionary order.

**b)** Give the three next permutations of permutation (1, 2, 3, 5, 8, 7, 6, 4) in dictionary order.

**c)** Build a generate algorithm enumerating all strings of $n$ characters, each character is taken from the set {a, b} that do not consisting two consecutive letter a.

## CONTENTS

1. Introduction to problem
2. Generating algorithm
3. **Backtracking algorithm**

## Backtracking (Thuật toán quay lui)

**3.1. Algorithm diagram**

3.2. Generate basic combinatorial configurations

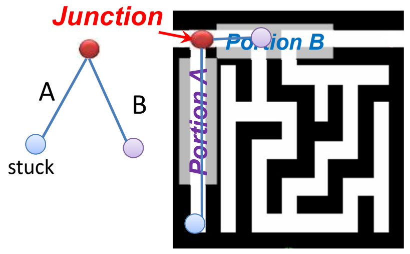
- Generate binary strings of length $n$
- Generate $m$-element subsets of the set of $n$ elements
- Generate permutations of $n$ elements

## Backtracking idea

- A clever form of exhaustive search.
- Backtracking is a technique used to solve problems with a large search space, by systematically trying and eliminating possibilities.

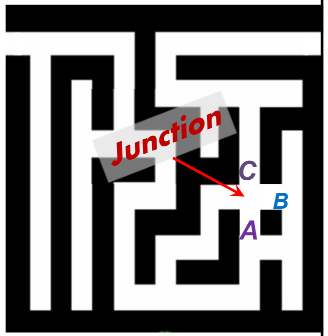
Example of backtracking would be going through a maze.

- At some point in a maze, you might have two options of which direction to go:
  - One strategy would be to try going through **Portion A** of the maze.
    - If you get stuck before you find your way out, then you ***"backtrack"*** to the junction.
  - At this point in time you know that **Portion A** will ***NOT*** lead you out of the maze,
    - so you then start searching in **Portion B**

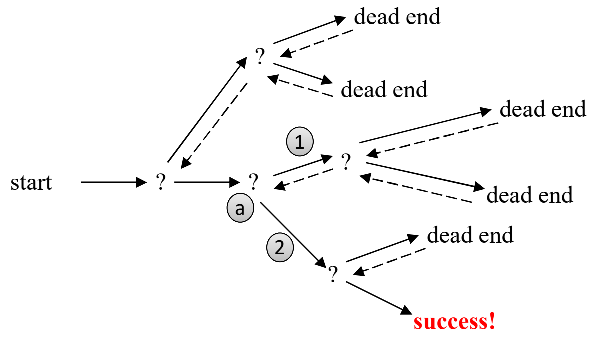


## Backtracking idea

- Clearly, at a single junction you could have even more than 2 choices.
- The backtracking strategy says to try each choice, one after the other,
  - if you ever get stuck, ***"backtrack"*** to the junction and try the next choice.
- If you try all choices and never found a way out, then there IS no solution to the maze.



## Backtracking (animation)



**Pseudo code for recursive backtracking algorithm:**

```
Backtrack(S)
  If S is a complete solution, report success
  For ( every possible choice e from current state / node) ⓐ
    If (S∪{e}) is not go to a dead end then Backtrack(S∪{e})  ①  ②
    Back out of the current choice to restore the state at the beginning of the loop. ⓐ
```

## Backtracking idea

- Dealing with the maze:
  - From your start point, you will iterate through each possible starting move.
  - From there, you recursively move forward.
  - If you ever get stuck, the recursion takes you back to where you were, and you try the next possible move.
- Make sure you don't try too many possibilities,
  - Mark which locations in the maze have been visited already so that no location in the maze gets visited twice.
  - If a place has already been visited, there is no point in trying to reach the end of the maze from there again.

## Backtracking diagram

- ***Enumeration problem* (*Q*):** *Given $A_1, A_2, \ldots, A_n$ be finite sets. Denote*

  $$A = A_1 \times A_2 \times \ldots \times A_n = \{ (a_1, a_2, \ldots, a_n):\ a_i \in A_i,\ i=1, 2, \ldots, n\}.$$

  *Assume P is a property on the set A. The problem is to enumerate all elements of the set A that satisfies the property P:*

  $$D = \{ a = (a_1, a_2, \ldots, a_n) \in A:\ a \text{ satisfy property } P \}.$$

- Elements of the set $D$ are called **feasible solution** (*lời giải chấp nhận được).*

## Backtracking diagram

All basic combinatorial enumeration problem could be rephrased in the form of Enumeration problem (Q).

Example:

- The problem of enumerating all binary string of length $n$ leads to the enumeration of elements of the set:

  $$B^n = \{(a_1, \ldots, a_n): a_i \in \{0, 1\}, i=1, 2, \ldots, n\}.$$

- The problem of enumerating all $m$-element subsets of set $N = \{1, 2, \ldots, n\}$ requires to enumerate elements of the set:

  $$S(m,n) = \{(a_1,\ldots, a_m) \in N^m: 1 \le a_1 < \ldots < a_m \le n \}.$$

- The problem of enumerating all permutations of natural numbers 1, 2, ..., $n$ requires to enumerate elements of the set

  $$\Pi_n = \{(a_1,\ldots, a_n) \in N^n:\ a_i \ne a_j ; i \ne j \}.$$

## Partial solution (Lời giải bộ phận)

The solution to the problem is an ordered tuple of $n$ elements $(a_1, a_2, \ldots, a_n)$, where $a_i \in A_i$, $i = 1, 2, \ldots, n$.

**Definition.** The $k$-level partial solution ($0 \le k \le n$) is an ordered tuple of $k$ elements

$$(a_1, a_2, \ldots, a_k),$$

where $a_i \in A_i$, $i = 1, 2, \ldots, k$.

- When $k = 0$, 0-level partial solution is denoted as ( ), and called as the empty solution.
- When $k = n$, we have a complete solution to a problem.

## Backtracking diagram

Backtracking algorithm is built based on the construction each component of solution one by one.

- Algorithm starts with empty solution ( ).
- Based on the property $P$, we determine which elements of set $A_1$ could be selected as the first component of solution. Such elements are called as **candidates** for the first component of solution. Denote candidates for the first component of solution as $S_1$. Take an element $a_1 \in S_1$, insert it into empty solution, we obtain 1-level partial solution: $(a_1)$.

- ***Enumeration problem* (*Q*):** *Given $A_1, A_2, \ldots, A_n$ be finite sets. Denote*

  $$A = A_1 \times A_2 \times \ldots \times A_n = \{ (a_1, a_2, \ldots, a_n):\ a_i \in A_i,\ i=1, 2, \ldots, n\}.$$

  *Assume P is a property on the set A. The problem is to enumerate all elements of the set A that satisfies the property P:*

  $$D = \{ a = (a_1, a_2, \ldots, a_n) \in A:\ a \text{ satisfy property } P \}.$$

## Backtracking diagram

- General step: Assume we have $k$-1 level partial solution:

$$(a_1, a_2, ..., a_{k-1}),$$

Now we need to build $k$-level partial solution:

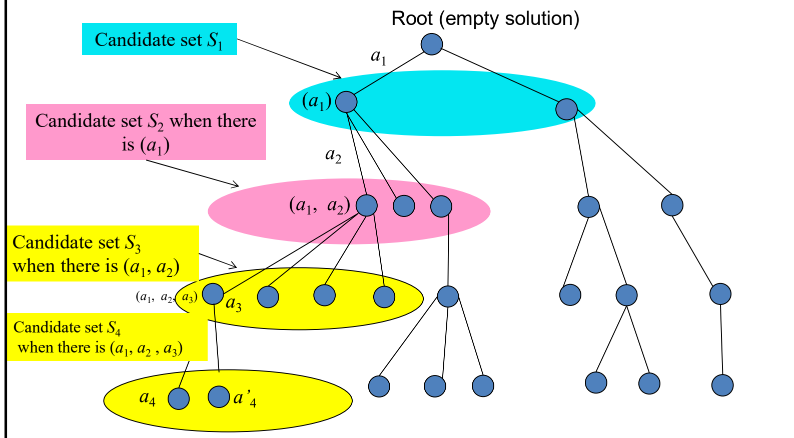
$$(a_1, a_2, ..., a_{k-1}, \mathbf{a_k})$$

  - Based on the property P, we determine which elements of set $A_k$ could be selected as the $k^{th}$ component of solution.
  - Such elements are called as candidates for the $k^{th}$ position of solution when $k$-1 first components have been chosen as $(a_1, a_2, ..., a_{k-1})$. Denote these candidates by $S_k$.
  - Consider 2 cases:
    - $S_k \neq \emptyset$
    - $S_k = \emptyset$

## Backtracking diagram

- **$S_k \neq \emptyset$:** Take $a_k \in S_k$ to insert it into current ($k$-1)-level partial solution $(a_1, a_2, ..., a_{k-1})$, we obtain $k$-level partial solution $(a_1, a_2, ..., a_{k-1}, a_k)$. Then
  - If $k = n$, then we obtain a complete solution to the problem,
  - If $k < n$, we continue to build the $(k+1)^{th}$ component of solution.
- **$S_k = \emptyset$:** It means the partial solution $(a_1, a_2, ..., a_{k-1})$ can not continue to develop into the complete solution. In this case, we **backtrack** to find new candidate for $(k\text{-}1)^{th}$ position of solution (note: this new candidate must be an element of $S_{k-1}$)
  - If one could find such candidate, we insert it into $(k\text{-}1)^{th}$ position, then continue to build the $k^{th}$ component.
  - If such candidate could not be found, we backtrack one more step to find new candidate for $(k\text{-}2)^{th}$ position,… If backtrack till the empty solution, we still can not find new candidate for 1st position, then the algorithm is finished.

## Decision tree for backtracking


Root (empty solution)
Candidate set $S_1$
$a_1$
$(a_1)$
Candidate set $S_2$ when there is $(a_1)$
$a_2$
$(a_1, a_2)$
Candidate set $S_3$ when there is $(a_1, a_2)$
$(a_1, a_2, a_3)$
$a_3$
Candidate set $S_4$ when there is $(a_1, a_2, a_3)$
$a_4$
$a'_4$

## Backtracking algorithm (recursive)

```
void Try(int $k$)
{
   <Build $S_k$ as the set to consist of candidates for the $k^{th}$
     component of solution>;
  for $y \in S_k$  //Each candidate $y$ of $S_k$
  {
     $a_k = y$;
     if ($k == n$) then <Record ($a_1, a_2, ..., a_k$) as a
     complete solution >;
     else Try($k$+1);
     Return the variables to their old states;
  }
}
```

**The call to execute backtracking algorithm: Try(1);**

- If only one solution need to be found, then it is necessary to find a way to terminate the nested recursive calls generated by the call to Try(1) once the first solution has just been recorded.
- If at the end of the algorithm, we obtain no solution, it means that the problem does not have any solution.

## Backtracking algorithm (not recursive)

```
void Backtracking ( )
{
 $k$=1;
 <Build $S_k$>;
 while ($k > 0$) {
   while ($S_k \neq \emptyset$ ) {
      $a_k \leftarrow S_k$; // Take $a_k$ from $S_k$
      if <($k == n$)> then <Record ($a_1,a_2,...,a_k$) as a
                                complete solution >;
      else {
           $k = k+1$;
           <Build $S_k$>;
      }
   }
   $k = k - 1$;  // Backtracking
 }
}
```

**The call to execute backtracking algorithm: Backtracking ( );**

## Two key issues

- In order to implement a backtracking algorithm to solve a specific combinatorial problems, we need to solve the following two basic problems:
  - Find algorithms to build candidate sets $S_k$
  - Find a way to describe these sets so that you can implement the operation to enumerate all their elements (implement the loop `for y ∈ S`$_k$).
    - The efficiency of the enumeration algorithm depends on whether we can accurately identify these candidate sets.

## Note

- If the length of complete solution is not known in advanced, and solutions are not needed to have the same length:

```
void Try(int k)
{
   <Build S_k as the set to consist of candidates for the k^th component of solution>;
  for y ∈ S_k  //Each candidate y of S_k
  {
     a_k = y;
     if  (k == n) then <Record (a_1, a_2, ..., a_k) as a complete solution >;
     else Try(k+1);
     Return the variables to their old states;
  }
}
```

- Then we need to modify statement

  **if (*k* == *n*) then <Record ($a_1$, $a_2$, ..., $a_k$) as a complete solution >;**
  **else Try(k+1);**

to

**if <($a_1$, $a_2$, ..., $a_k$) is a complete solution> then <Record ($a_1$, $a_2$, ..., $a_k$) as a complete solution >;**
**else Try(k+1);**

→ Need to build a function to check whether $(a_1, a_2, ..., a_k)$ is the complete solution.

## Backtracking (Thuật toán quay lui)

3.1. Algorithm diagram

3.2. Generate basic combinatorial configurations

- **Generate binary strings of length *n***
- Generate *m*-element subsets of the set of *n* elements
- Generate permutations of *n* elements

## Example 1: Enumerate all binary string of length *n*

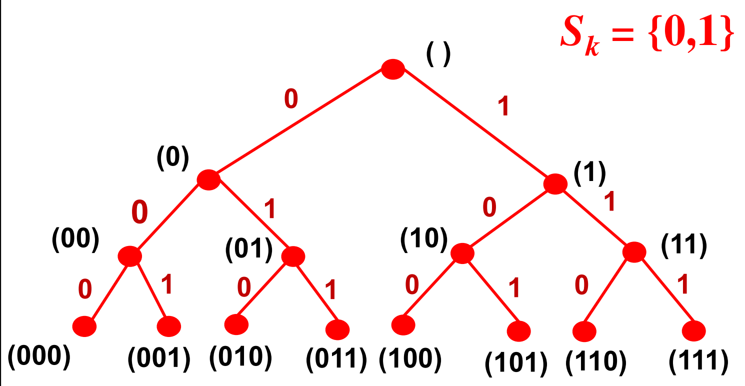
- Problem to enumerate all binary string of length *n* leads to the enumeration of all elements of the set:

  $A^n = \{(a_1, ..., a_n): a_i \in \{0, 1\}, i=1, 2, ..., n\}.$

- We consider how to solve two issue keys to implement backtracking algorithm:
  - Build candidate set $S_k$: We have $S_1 = \{0, 1\}$. Assume we have binary string of length *k*-1 $(a_1, ..., a_{k-1})$, then $S_k = \{0,1\}$. Thus, the candidate sets for each position of the solution are determined.
  - Implement the loop to enumerate all elements of $S_k$: we can use the loop `for`

```
for y in range (0, 2)
    //OR: for (y=0; y<=1; y++) in C/C++
```

## Decision tree to enumerate binary strings of length 3

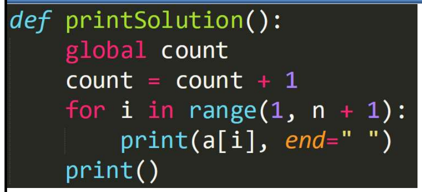


## Program in C++ (Recursive)

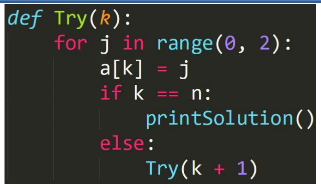
```
#include <iostream>
using namespace std;
int  n, count;
int a[100];

void PrintSolution()
{
  int i, j;
  count++;
  cout<<"String # " <<count<<": ";
  for  (i=1 ; i<= n ;i++)
  {
       j=a[i];
       cout<<j<<"   ";
  }
  cout<<endl;
}
```

```
void  Try(int k)
{
   for (int j = 0; j<=1; j++)
   {
       a[k] = j;
       if (k == n)  PrintSolution();
       else  Try(k+1);
    }
}

int main()
{
   cout<<"Enter n = ";cin>>n;
   count = 0; Try(1);
   cout<<"Number of strings "<<count;
}
```

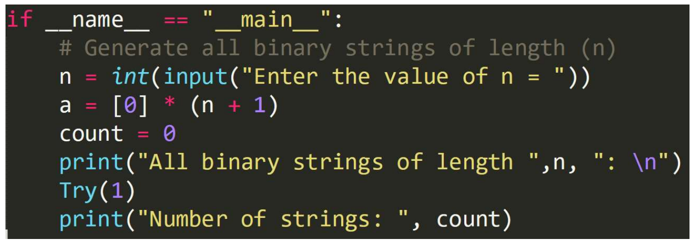
## Program in Python (Recursive)



```
def printSolution():
    global count
    count = count + 1
    for i in range(1, n + 1):
        print(a[i], end=" ")
    print()
```

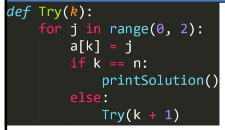


```
def Try(k):
    for j in range(0, 2):
        a[k] = j
        if k == n:
            printSolution()
        else:
            Try(k + 1)
```

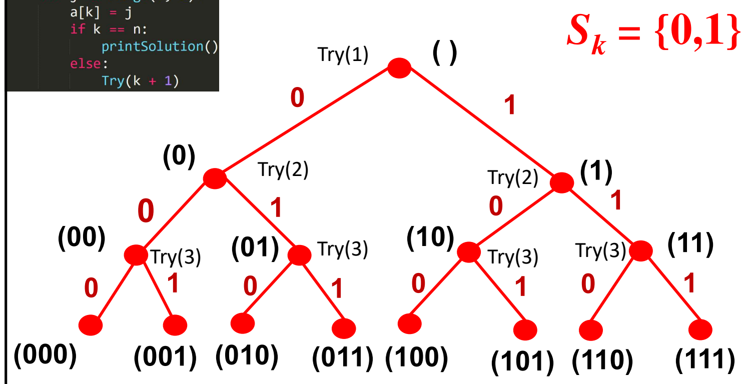


```
if __name__ == "__main__":
    # Generate all binary strings of length (n)
    n = int(input("Enter the value of n = "))
    a = [0] * (n + 1)
    count = 0
    print("All binary strings of length ",n, ": \n")
    Try(1)
    print("Number of strings: ", count)
```

## Decision tree to enumerate binary strings of length 3



```
def Try(k):
    for j in range(0, 2):
        a[k] = j
        if k == n:
            printSolution()
        else:
            Try(k + 1)
```



## Program in C++ (non recursive)

```
#include <iostream>
using namespace std;
int  n, count,k;
int a[100], s[100];

void PrintSolution()
{
    int i, j;
    count++;
    cout<<"String # " << count<<": ";
    for  (i=1 ; i<= n ;i++)
       cout<<a[i]<<"  ";

     cout<<endl;
}
```

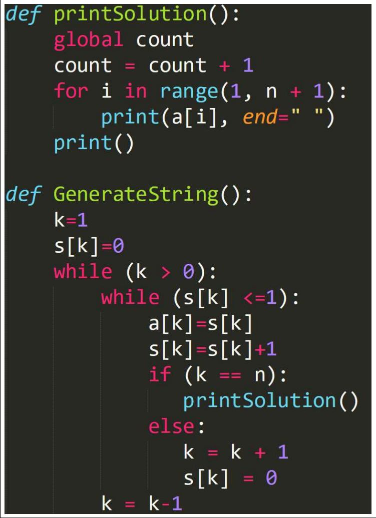
```
void  GenerateString( )
{
     k=1;  s[k]=0;
     while (k > 0)
     {
         while (s[k] <= 1)
         {
              a[k]=s[k];
              s[k]=s[k]+1;
              if (k==n)  PrintSolution();
              else
              {
                   k++; s[k]=0;
               }
          }
          k--;   // BackTrack
     }
}
```

## Program in C++ (non recursive)

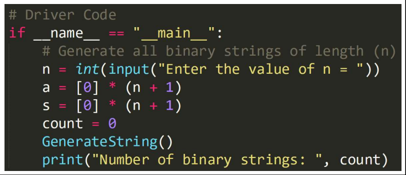
```
int main() {
      cout<<"Enter value of  n = ";cin>>n;
      count = 0; GenerateString();
      cout<<"Number of strings = "<<count<<endl;

}
```

NGUYỄN KHÁNH PHƯƠNG
CS-SOICT-HUST

## Program in Python (non recursive)



```
def printSolution():
    global count
    count = count + 1
    for i in range(1, n + 1):
        print(a[i], end=" ")
    print()

def GenerateString():
    k=1
    s[k]=0
    while (k > 0):
        while (s[k] <=1):
            a[k]=s[k]
            s[k]=s[k]+1
            if (k == n):
               printSolution()
            else:
               k = k + 1
               s[k] = 0
        k = k-1
```



```
# Driver Code
if __name__ == "__main__":
    # Generate all binary strings of length (n)
    n = int(input("Enter the value of n = "))
    a = [0] * (n + 1)
    s = [0] * (n + 1)
    count = 0
    GenerateString()
    print("Number of binary strings: ", count)
```

## Backtracking (Thuật toán quay lui)

3.1. Algorithm diagram

3.2. Generate basic combinatorial configurations

- Generate binary strings of length *n*
- Generate *m*-element subsets of the set of *n* elements
- Generate permutations of *n* elements

NGUYỄN KHÁNH PHƯƠNG
CS-SOICT-HUST

## Example 2. Generate *m*-element subsets of the set of *n* elements

**Problem:** Enumerate all *m*-element subsets of the set *n* elements $N = \{1, 2, ..., n\}$.

Example: Enumerate all 3-element subsets of the set 5 elements N = {1, 2, 3, 4, 5}

Solution: (1, 2, 3), (1, 2, 4), (1, 2, 5), (1, 3, 4), (1, 3, 5), (1, 4, 5), (2, 3, 4), (2, 3, 5), (2, 4, 5), (3, 4, 5)

➔ Equivalent problem: Enumerate all elements of set:

$$S(m,n)=\{(a_1,\dots, a_m)\in N^m: 1 \le a_1<\dots<a_m\le n\}$$

## Example 2. Generate *m*-element subsets of the set of *n* elements

We consider how to solve two issue keys to implement backtracking:

- Build candidate set $S_k$:
  - With the condition: : $1 \le a_1 < a_2 < \dots < a_m \le n$

  we have $S_1 = \{1, 2, ..., n-(m-1)\}$.
  - Assume the current subset is $(a_1, ..., a_{k-1})$, with the condition $a_{k-1} < a_k < \dots < a_m \le n$, we have $S_k = \{a_{k-1}+1, a_{k-1}+2, ..., n-(m-k)\}$.
- Implement the loop to enumerate all elements of $S_k$: we can use the loop `for`

```
for j in range (a[k-1]+1, n-m+k+1)
 //for (j=a[k-1]+1;j<=n-m+k;j++) …in C++
```

## Program in C++ (Recursive)

```
#include <iostream>
using namespace std;

int  n, m, count;
int a[100];
void PrintSolution() {
    int i;
    count++;
    cout<<"The subset #" <<count<<": ";
    for  (i=1 ; i<= m ;i++)
          cout<<a[i]<<"  ";
    cout<<endl;
}
```

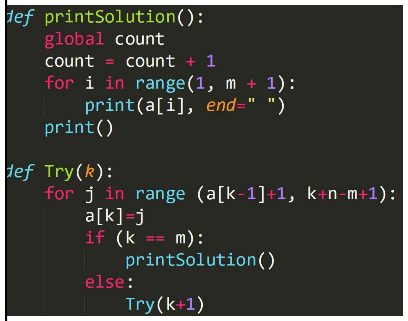
```
void  Try(int k){
   int j;
   for (j = a[k-1] +1; j<= n-m+k; j++) {
        a[k] = j;
        if (k==m)  PrintSolution();
        else Try(k+1);
   }
}
int main() {
     cout<<"Enter n, m = "; cin>>n; cin>>m;
     a[0]=0; count = 0; Try(1);
     cout<<"Number of "<<m<<"-element subsets of set "<<n<<" elements = "<<count<<endl;
}
```

## Program in C++ (Non Recursive)

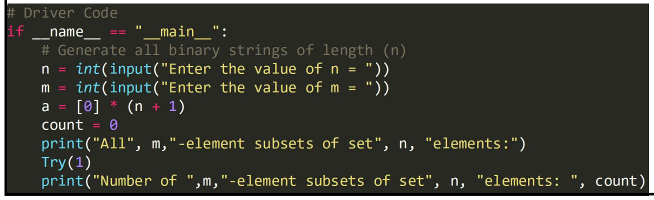
```
#include <iostream>
using namespace std;

int  n, m, count,k;
int a[100], s[100];
void PrintSolution() {
    int i;
    count++;
    cout<<"The subset #" <<count<<": ";
    for  (i=1 ; i<= m ;i++)
          cout<<a[i]<<"  ";
    cout<<endl;
}
```

```
void  MSet()
{
   k=1;  s[k]=1;
   while(k>0){
      while (s[k]<= n-m+k)  {
           a[k]=s[k];  s[k]=s[k]+1;
           if (k==m) PrintSolution();
           else { k++; s[k]=a[k-1]+1;  }
      }
      k--;
   }
}
int main() {
     cout<<"Enter n, m = "; cin>>n; cin>>m;
     a[0]=0; count = 0; MSet();
     cout<<"Number of "<<m<<"-element subsets of set "<<n<<" elements = "<<count<<endl;
}
```

## Program in Python (Recursive)

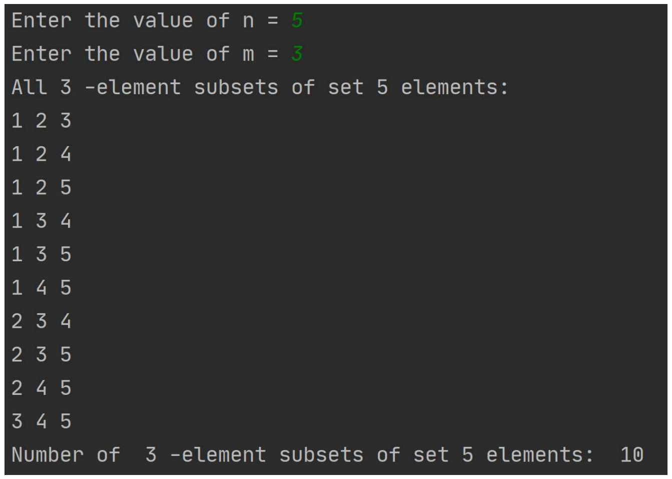


```
def printSolution():
    global count
    count = count + 1
    for i in range(1, m + 1):
        print(a[i], end=" ")
    print()

def Try(k):
    for j in range (a[k-1]+1, k+n-m+1):
        a[k]=j
        if (k == m):
            printSolution()
        else:
            Try(k+1)
```



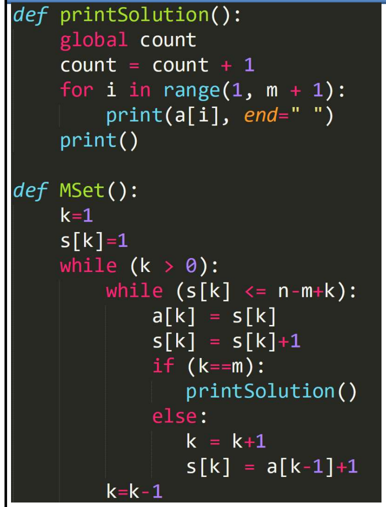
```
# Driver Code
if __name__ == "__main__":
    # Generate all binary strings of length (n)
    n = int(input("Enter the value of n = "))
    m = int(input("Enter the value of m = "))
    a = [0] * (n + 1)
    count = 0
    print("All", m,"-element subsets of set", n, "elements:")
    Try(1)
    print("Number of ",m,"-element subsets of set", n, "elements: ", count)
```

## Program in Python (Recursive)



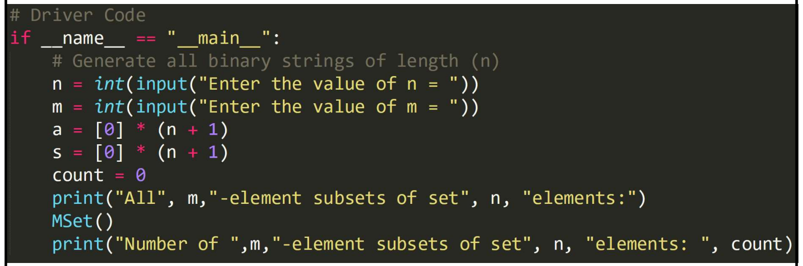
```
Enter the value of n = 5
Enter the value of m = 3
All 3 -element subsets of set 5 elements:
1 2 3
1 2 4
1 2 5
1 3 4
1 3 5
1 4 5
2 3 4
2 3 5
2 4 5
3 4 5
Number of  3 -element subsets of set 5 elements:  10
```

## Program in Python (Non Recursive)



```
def printSolution():
    global count
    count = count + 1
    for i in range(1, m + 1):
        print(a[i], end=" ")
    print()

def MSet():
    k=1
    s[k]=1
    while (k > 0):
        while (s[k] <= n-m+k):
            a[k] = s[k]
            s[k] = s[k]+1
            if (k==m):
                printSolution()
            else:
                k = k+1
                s[k] = a[k-1]+1
        k=k-1
```

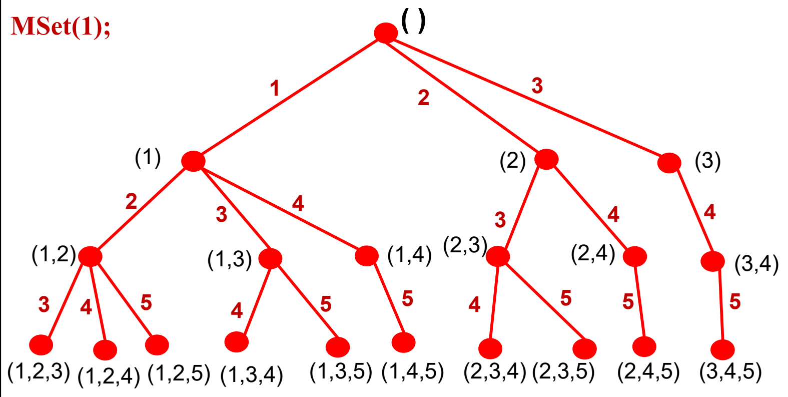
## Program in Python (Non Recursive)



```
# Driver Code
if __name__ == "__main__":
    # Generate all binary strings of length (n)
    n = int(input("Enter the value of n = "))
    m = int(input("Enter the value of m = "))
    a = [0] * (n + 1)
    s = [0] * (n + 1)
    count = 0
    print("All", m,"-element subsets of set", n, "elements:")
    MSet()
    print("Number of ",m,"-element subsets of set", n, "elements: ", count)
```

## Decision tree S(5,3)

**MSet(1);**



$S_k = \{a_{k-1}+1, a_{k-1}+2, ..., n-(m-k)\}$

## Backtracking (Thuật toán quay lui)

3.1. Algorithm diagram

3.2. Generate basic combinatorial configurations

- Generate binary strings of length $n$
- Generate $m$-element subsets of the set of $n$ elements
- **Generate permutations of $n$ elements**

NGUYỄN KHÁNH PHƯƠNG
CS-SOICT-HUST

## Example 3. Enumerate permutations

Permutation set of natural numbers 1, 2, ..., $n$ is the set:

$$\Pi_n = \{(x_1,..., x_n) \in N^n: \ x_i \neq x_j \ , \ i \neq j \ \}.$$

**Problem: Enumerate all elements of $\Pi_n$**

NGUYỄN KHÁNH PHƯƠNG
CS-SOICT-HUST

## Example 3. Enumerate permutations

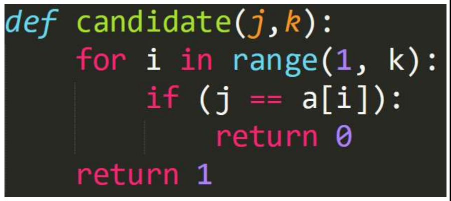
- Build candidate set $S_k$:
  - Actually $S_1 = N$. Assume we have current partial permutation $(a_1, a_2, ..., a_{k-1})$, with the condition $a_i \neq a_j$, for all $i \neq j$ , we have

    $S_k = N \setminus \{ \ a_1, a_2, ..., a_{k-1} \}.$

NGUYỄN KHÁNH PHƯƠNG
CS-SOICT-HUST

## Describe $S_k$

**Build function to detect candidates:**

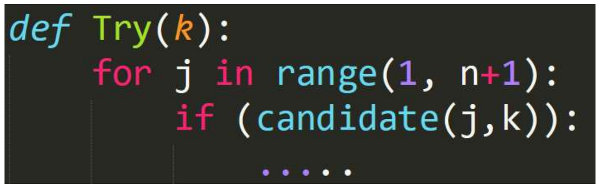


```python
def candidate(j,k):
    for i in range(1, k):
        if (j == a[i]):
            return 0
    return 1
```

```
bool candidate(int j, int k)
{
  //function returns true if and only if j ∊ Sk
   int i;
   for (i=1;i++;i<=k-1)
       if (j == a[i]) return false;
   return true;
}
```

## Example 3. Enumerate permutations

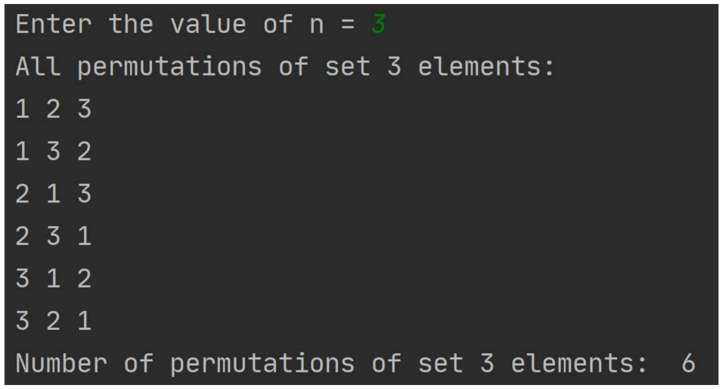
- Implement the loop to enumerate all elements of $S_k$:



```python
def Try(k):
    for j in range(1, n+1):
        if (candidate(j,k)):
            .....
```

```
for (j=1; j <= n; j++;)
  if  (candidate(j, k))
  {
    // j is candidate for position kth
     . . .
   }
```

## Example 3. Enumerate permutations



```
Enter the value of n = 3
All permutations of set 3 elements:
1 2 3
1 3 2
2 1 3
2 3 1
3 1 2
3 2 1
Number of permutations of set 3 elements:  6
```

## Program in C++ (Recursive)

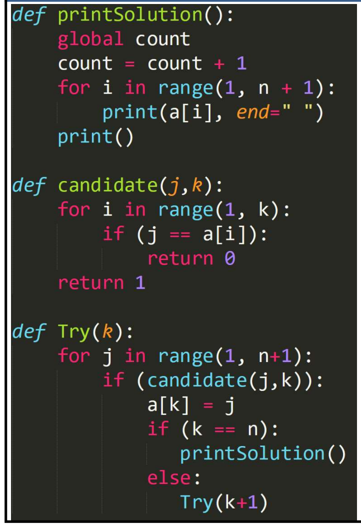
```
#include <iostream>
using namespace std;

int  n, m, count;
int a[100];

int PrintSolution() {
    int i, j;
    count++;
    cout<<"Permutation #"<<count<<": ";
    for  (i=1 ; i<= n ;i++)
         cout<<a[i]<<"  ";
    cout<<endl;
}
```

```
bool candidate(int j, int k)
{
    int i;
    for (i=1; i<=k-1; i++)
       if (j == a[i])
              return false;
    return true;
}
```

## Program in C++ (Recursive)

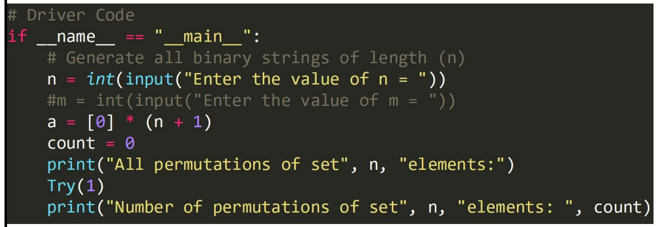
```
void  Try(int k)
{
   int j;
   for (j = 1; j<=n; j++)
      if (candidate(j,k))
      {  a[k] = j;
         if (k==n)  PrintSolution( );
         else  Try(k+1);
      }
}
int main() {
    cout<<(“Enter n = "); cin>>n;
    count = 0; Try(1);
    cout<<“Number of permutations = " << count;
}
```

## Program in Python (Recursive)

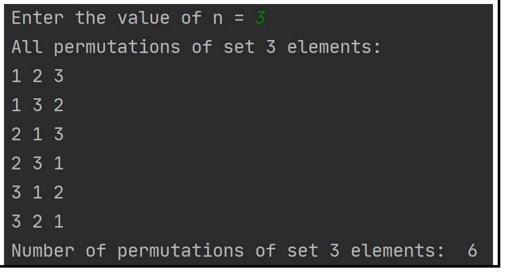


```
def printSolution():
    global count
    count = count + 1
    for i in range(1, n + 1):
        print(a[i], end=" ")
    print()

def candidate(j,k):
    for i in range(1, k):
        if (j == a[i]):
            return 0
    return 1

def Try(k):
    for j in range(1, n+1):
        if (candidate(j,k)):
            a[k] = j
            if (k == n):
                printSolution()
            else:
                Try(k+1)
```

## Program in Python (Recursive)

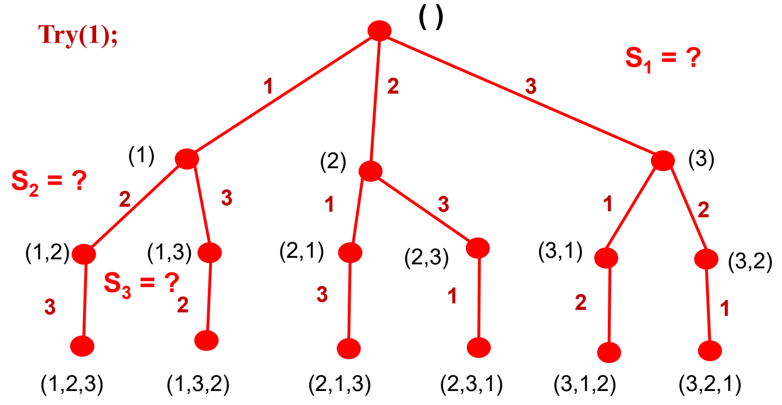


```
# Driver Code
if __name__ == "__main__":
    # Generate all binary strings of length (n)
    n = int(input("Enter the value of n = "))
    #m = int(input("Enter the value of m = "))
    a = [0] * (n + 1)
    count = 0
    print("All permutations of set", n, "elements:")
    Try(1)
    print("Number of permutations of set", n, "elements: ", count)
```



```
Enter the value of n = 3
All permutations of set 3 elements:
1 2 3
1 3 2
2 1 3
2 3 1
3 1 2
3 2 1
Number of permutations of set 3 elements:  6
```

## Decision tree to enumerate permutations of 1, 2, 3



$$S_k = N \setminus \{ a_1, a_2, ..., a_{k-1}\}$$