TRƯỜNG ĐẠI HỌC BÁCH KHOA HÀ NỘI
VIỆN CÔNG NGHỆ THÔNG TIN VÀ TRUYỀN THÔNG

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

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.

NGUYỄN KHÁNH PHƯƠNG
Bộ môn KHMT – ĐHBK HN

TRƯỜNG ĐẠI HỌC BÁCH KHOA HÀ NỘI
VIỆN CÔNG NGHỆ THÔNG TIN VÀ TRUYỀN THÔNG

Chapter 3

ENUMERATION ROBLEM

NGUYỄN KHÁNH PHƯƠNG
Bộ môn KHMT – ĐHBK HN

CONTENTS

NGUYỄN KHÁNH PHƯƠNG
Bộ môn KHMT – ĐHBK HN

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

NGUYỄN KHÁNH PHƯƠNG
Bộ môn KHMT – ĐHBK HN

Introduction to problem

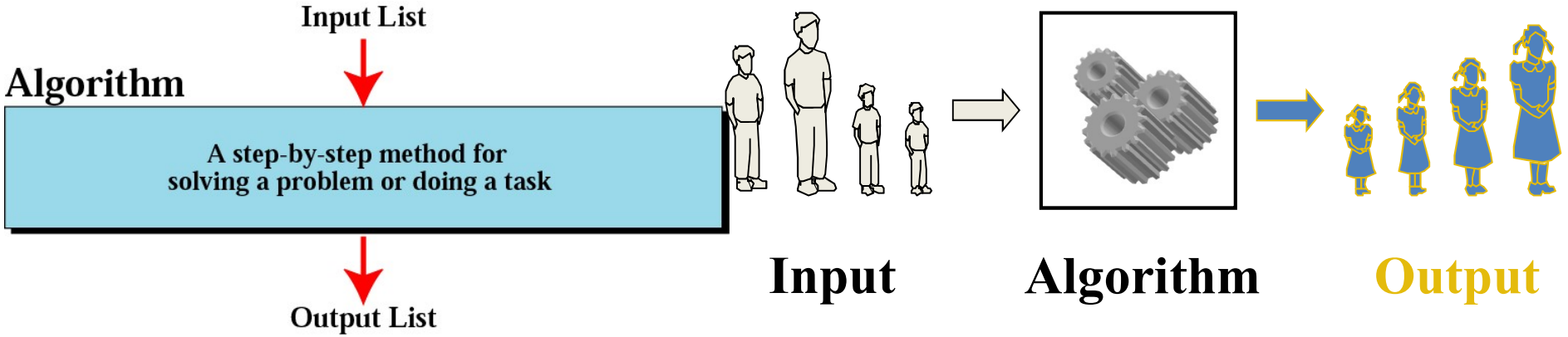
- The combinatorial enumeration problem is solvable if we can define ***an algorithm*** so that all configurations could be built in turn.
- An enumeration algorithm must satisfy 2 basic requirements:
  - Do not repeat a configuration,
  - Do not miss a configuration.

CONTENTS

NGUYỄN KHÁNH PHƯƠNG
Bộ môn KHMT – ĐHBK HN

Algorithm

- The word *algorithm* comes from the name of a Persian mathematician Abu Ja'far Mohammed ibn-i Musa al Khowarizmi.
- In computer science, this word refers to a special method consisting of a sequence of unambiguous instructions useable by a computer for solution of a problem.
- Informal definition of an algorithm in a computer:



- Example: The problem of finding the largest integer among a number of positive integers
  - Input: the array of n positive integers $a_1, a_2, \ldots, a_n$
  - Output: the largest
  - Example: Input 12 8 13 9 11 ➔ Output: 12
  - Question: Design the algorithm to solve this problem

Algorithm

- All algorithms must satisfy the following criteria:

  (1) **Input**. The algorithm receives data from a certain set.

  (2) **Output**. For each set of input data, the algorithm gives the solution to the problem.

  (3) **Precision**. Each instruction is clear and unambiguous.

  (4) **Finiteness**. If we trace out the instructions of an algorithm, then for all cases, the algorithm terminates after a finite (possibly very large) number of steps.

  (5) **Uniqueness**. The intermediate results of each step of the algorithm are uniquely determined and depend only on the input and the result of the previous steps.

  (6) **Generality.** The algorithm could be applied to solve any problem with a given form

Comparing Algorithms

- Given 2 or more algorithms to solve the same problem, how do we select the best one?
- Some criteria for selecting an algorithm:
  1) Is it easy to implement, understand, modify?
  2) How long does it take to run it to completion? **TIME**
  3) How much of computer memory does it use? **SPACE**

In this lecture we are interested in the second and third criteria:

- **Time complexity**: The amount of time that an algorithm needs to run to completion
- **Space complexity**: The amount of memory an algorithm needs to run

We will occasionally look at space complexity, but we are mostly interested in time complexity in this course. Thus in this course the better algorithm is the one which runs faster (has smaller time complexity)

How to Calculate Running time

- Most algorithms transform input objects into output objects

The running time of an algorithm typically grows with the input size

- Idea: analyze running time as a function of input size
- Even on inputs of the same size, running time can be very different

Example: algorithm that finds the first prime number in an array by scanning it left to right

- Array 1: 3 9 8 12 15 20 (the algorithm stops once the 1st element is considered)
- Array 2: 9 8 3 12 15 20 (the algorithm stops once the 3rd element is considered)
- Array 3: 9 8 12 15 20 3 (the algorithm stops until the last element is considered)

➔ Idea: analyze running time in the

  - best case
  - worst case
  - average case

Kind of analyses

Best-case:

- T(n) = minimum time required to execute the algorithm on any input of size *n*
- Treat with a slow algorithm that works fast on some input.

Average-case: (sometimes)

- T(*n*) = expected average time of algorithm over all inputs of size *n*.
- Need assumption of statistical distribution of inputs
- Very useful but often difficult to determine

Worst-case: (usually)

- T(*n*) = maximum time of algorithm on any input of size *n*.
- Easier to analyze

To evaluate the running time: 2 ways:

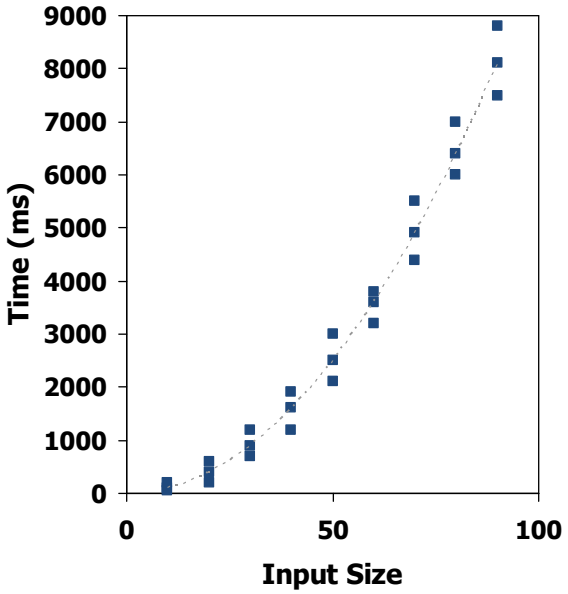
- Experimental evaluation of running time
- Theoretical analysis of running time

Experimental Evaluation of Running Time

- Write a program implementing the algorithm
- Run the program with inputs of varying size and composition
- Use a method like clock( ) to get an accurate measure of the actual running time

```
clock_t startTime = clock();
doSomeOperation();
clock_t endTime = clock();
clock_t clockTicksTaken = endTime - startTime;
double timeInSeconds = clockTicksTaken / (double) CLOCKS_PER_SEC;
```

- Plot the results



Limitations of Experiments when evaluating the running time of an algorithm

- Experimental evaluation of running time is very useful but
  - It is necessary to implement the algorithm, which may be difficult
  - Results may not be indicative of the running time on other inputs not included in the experiment
  - In order to compare two algorithms, the same hardware and software environments must be used

➔ We need: **Theoretical Analysis of Running Time**

Theoretical Analysis of Running Time

- Uses a pseudo-code description of the algorithm instead of an implementation
- Characterizes running time as a function of the input size, ***n***
- Takes into account all possible inputs
- Allows us to evaluate the speed of an algorithm independent of the hardware/software environment (Changing the hardware/software environment affects the running time by a constant factor, but does not alter the growth rate of the running time)

Asymptotic notation

Θ, Ω, O, **o**, ω

» What these symbols do are:

- give us a notation for talking about how fast a function goes to infinity, which is just what we want to know when we study the running times of algorithms.
- defined for functions over the natural numbers
- used to compare the order of growth of 2 functions

Example: $f(n) = \Theta(n^2)$: Describes how $f(n)$ grows in comparison to n^2.

» Instead of working out a complicated formula for the exact running time, we can just say that the running time is for example $\Theta(n^2)$ [read as theta of n^2]: that is, the running time is proportional to n^2 plus lower order terms. For most purposes, that's just what we want to know.

Θ - Theta notation

- For a given function $g(n)$, we denote by $\boldsymbol{\Theta}(g(n))$ the set of functions

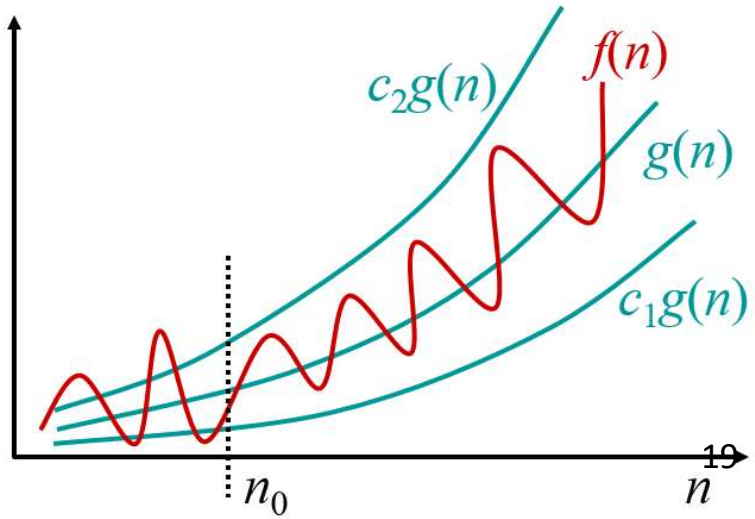
$$\Theta(g(n)) = \left\{ \begin{array}{c} f(n) : \text{there exist positive constants } c_1, c_2, \text{and } n_0 \text{ s.t.} \\ 0 \le c_1 g(n) \le f(n) \le c_2 g(n) \text{ for all } n \ge n_0 \end{array} \right\}$$

Intuitively: Set of all functions that have the same *rate of growth* as $g(n)$.

- A function $f(n)$ belongs to the set $\boldsymbol{\Theta}(g(n))$ if there exist positive constants c_1 and c_2 such that it can be "sand- wiched" between $c_1 g(n)$ and $c_2 g(n)$ for sufficiently large n
  - $f(n) = \boldsymbol{\Theta}(g(n))$ means that there exists some constant c_1 and c_2 s.t.

$$c_1 g(n) \le f(n) \le c_2 g(n) \text{ for large enough } n.$$

- When we say that one function is theta of another, we mean that neither function goes to infinity faster than the other.



$f(n) = \Theta(g(n)) \iff \exists c_1, c_2, n_0 > 0 : \forall n \geq n_0, \; c_1 g(n) \leq f(n) \leq c_2 g(n)$

Example 1: Show that $10n^2 - 3n = \mathbf{\Theta}(n^2)$

- With which values of the constants n_0, c_1, c_2 then the inequality in the definition of the theta notation is correct:

$$c_1 n^2 \leq f(n) = 10n^2 - 3n \leq c_2 n^2 \; \forall n \geq n_0$$

- Suggestion: Make c_1 a little smaller than the leading (the highest) coefficient, and c_2 a little bigger.

➔ Select: $c_1 = 1$, $c_2 = 11$, $n_0 = 1$ then we have

$$n^2 \leq 10n^2 - 3n \leq 11n^2, \; with \; n \geq 1.$$

➔ $\forall n \geq 1$: $10n^2 - 3n = \mathbf{\Theta}(n^2)$

- Note: For polynomial functions: To compare the growth rate, it is necessary to look at the term with the highest coefficient

$f(n) = \Theta(g(n))$ $\Longleftrightarrow$ $\exists c_1, c_2, n_0 > 0 \ : \ \forall n \geq n_0, \ c_1 g(n) \leq f(n) \leq c_2 g(n)$

Example 2: Show that $f(n) = \frac{1}{2}n^2 - 3n = \boldsymbol{\Theta}(n^2)$

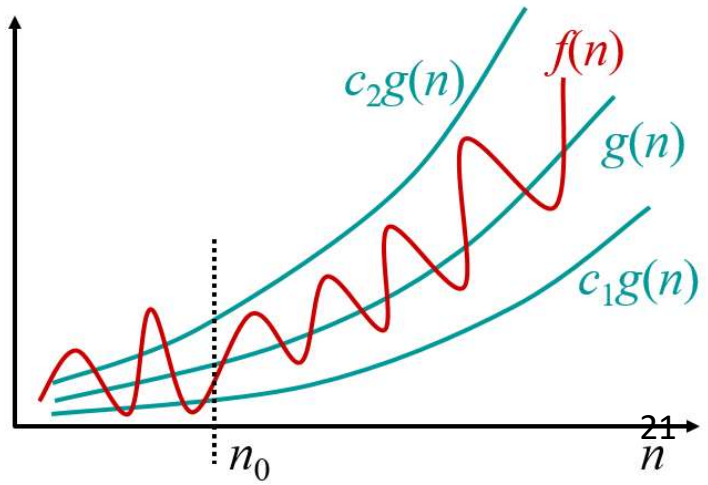
We must find n_0, c_1 and c_2 such that

$$c_1 n^2 \leq f(n) = \frac{1}{2}n^2 - 3n \leq c_2 n^2 \ \forall n \geq n_0$$

Select $c_1 = ¼$, $c_2 = 1$, and $n_0 = 7$ we have:

$$\frac{1}{4}n^2 \leq f(n) = \frac{1}{2}n^2 - 3n \leq n^2 \quad \forall\, n \geq 7$$

➔$\forall\, n \geq 7$: $\frac{1}{2}n^2 - 3n = \boldsymbol{\Theta}(n^2)$

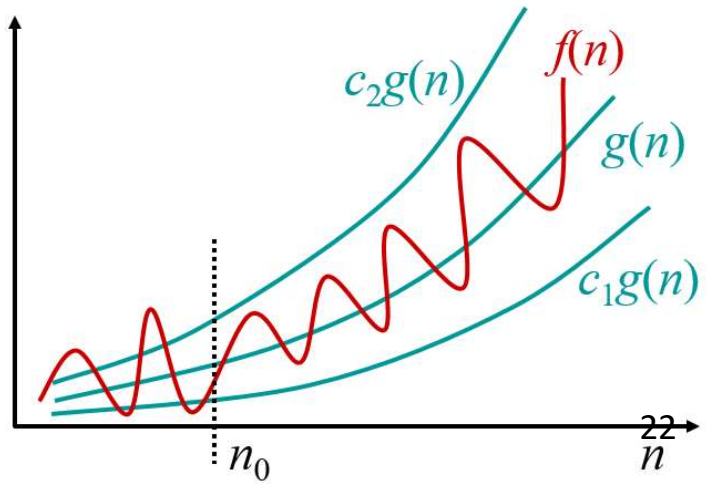


$f(n) = \Theta(g(n))$ $\iff$ $\exists c_1, c_2, n_0 > 0 \ : \ \forall n \geq n_0, \ c_1 g(n) \leq f(n) \leq c_2 g(n)$

Example 3: Show that $f(n) = 23n^3 - 10\, n^2 \log_2 n + 7n + 6 = \boldsymbol{\Theta}(n^3)$

We must find n_0, c_1 and c_2 such that

$$c_1 n^3 \leq f(n) = 23n^3 - 10\, n^2 \log_2 n + 7n + 6 \leq c_2 n^3 \forall n \geq n_0$$



$f(n) = \Theta(g(n)) \iff \exists c_1, c_2, n_0 > 0 \ : \ \forall n \geq n_0, \ c_1 g(n) \leq f(n) \leq c_2 g(n)$

Example 3: Show that $f(n) = 23n^3 - 10\, n^2 \log_2 n + 7n + 6 = \Theta(n^3)$

$f(n) = 23n^3 - 10\, n^2 \log_2 n + 7n + 6$

➔ $f(n) = [23 - (10 \log_2 n)/n + 7/n^2 + 6/n^3]n^3$

Then:

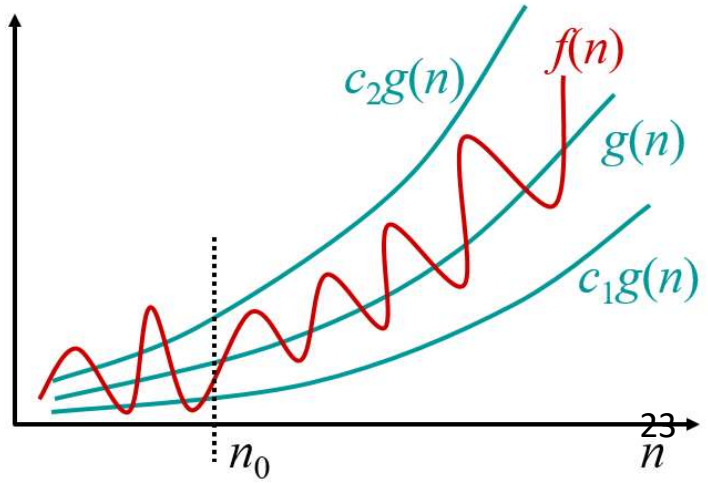
- $\forall n \geq 10 : \ f(n) \leq (23 + 0 + 7/100 + 6/1000)n^3$
  $= (23 + 0 + 0.07 + 0.006)n^3$
  $= 23.076\, n^3 < 24\, n^3$
- $\forall n \geq 10 : \ f(n) \geq (23 - \log_2 10 + 0 + 0)n^3 > (23 - \log_2 16)n^3 = 19n^3$

➔ We have:

$\forall n \geq 10: \ 19\, n^3 \leq f(n) \leq 24\, n^3$

$(n_0 = 10, \ c_1 = 19, \ c_2 = 24, \ g(n) = n^3)$

Therefore: $f(n) = \Theta(n^3)$



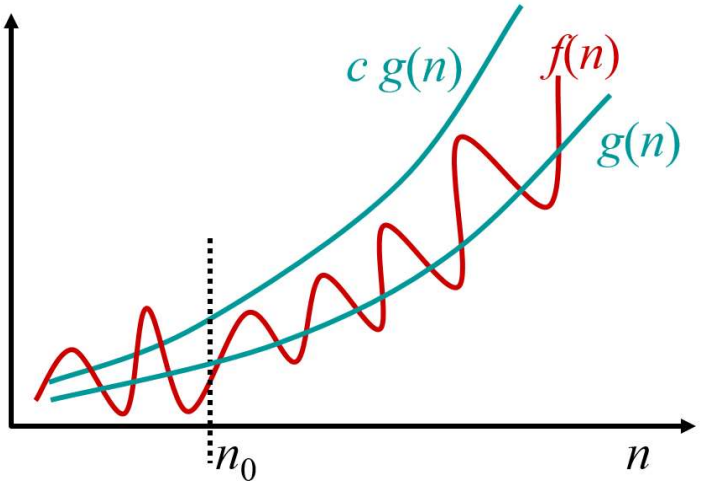
O - big Oh notation

For a given function $g(n)$, we denote by $O(g(n))$ the set of functions

$$O(g(n)) = \left\{ \begin{array}{c} f(n) : \text{there exist positive constants } c \text{ and } n_0 \text{ s.t.} \\ 0 \le f(n) \le cg(n) \text{ for all } n \ge n_0 \end{array} \right\}$$

Intuitively: Set of all functions whose *rate of growth* is the same as or lower than that of $g(n)$.

- We say: $g(n)$ is asymptotic upper bound of the function $f(n)$, to within a constant factor, and write $f(n) = O(g(n))$.
- $f(n) = O(g(n))$ means that there exists some constant c such that $f(n)$ is always $\le cg(n)$ for large enough n.
- $O(g(n))$ is the set of functions that go to infinity no faster than $g(n)$.



Graphic Illustration

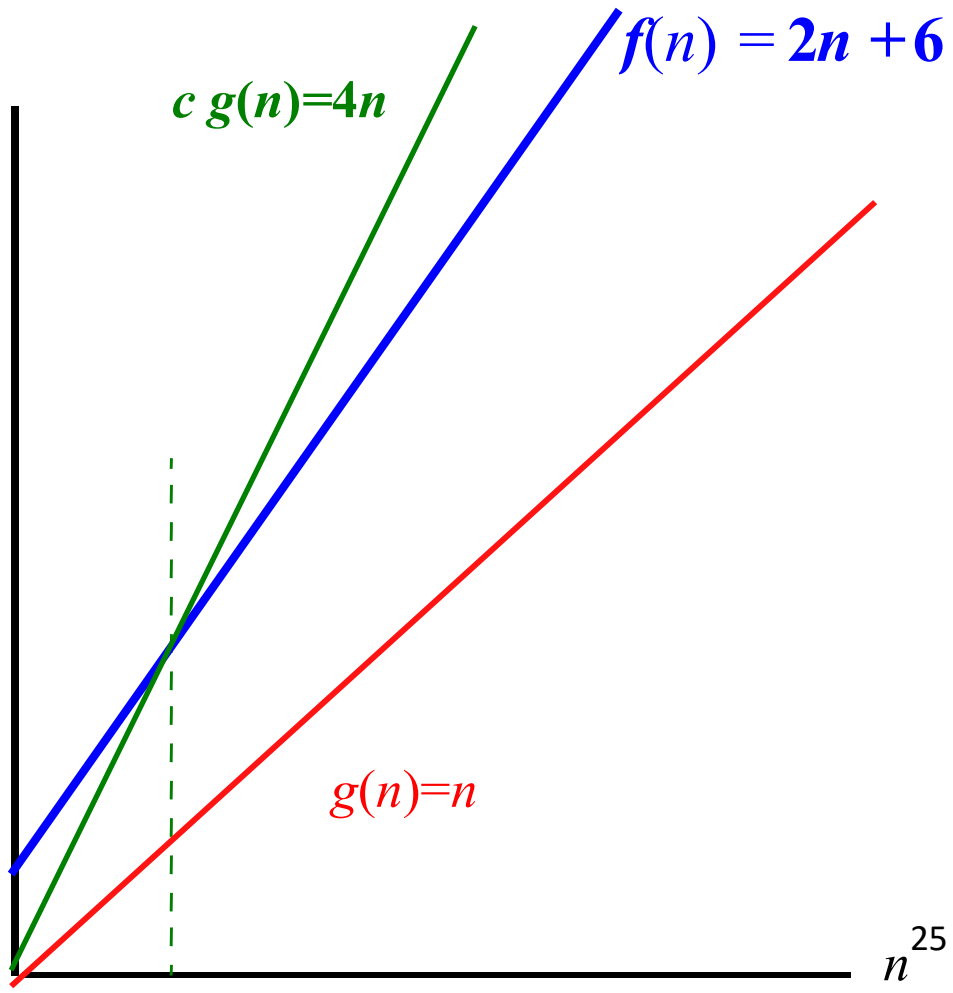
$O(g(n))$ = {$f(n)$: ∃ positive constants c and n_0, such that $\forall n \geq n_0$, we have $0 \leq f(n) \leq cg(n)$ }

- $f(n) = 2n+6$
- Conf. def:
  - Need to find a function $g(n)$ and constants c and n_0 such as $f(n) < cg(n)$ when $n > n_0$

➔ $g(n) = n$, $c = 4$ and $n_0 = 3$

→ $f(n)$ is $O(n)$

The order of $f(n)$ is n



Big-Oh Examples

$O(g(n)) = \{f(n) : \exists$ positive constants c and n_0, such that $\forall n \geq n_0$, we have $0 \leq f(n) \leq cg(n)\}$

- Example 1: Show that $2n + 10 = O(n)$

➔ $f(n) = 2n+10$, $g(n) = n$

  - Need constants c and n_0 such that $2n + 10 \leq cn$ for $n \geq n_0$
  - $(c - 2)\, n \geq 10$
  - $n \geq 10/(c - 2)$
  - Pick $c = 3$ and $n_0 = 10$

- Example 2: Show that $7n$-2 is $O(n)$

➔ $f(n) = 7n$-2, $g(n) = n$

  - Need constants c and n_0 such that $7n - 2 \leq cn$ for $n \geq n_0$
  - $(7 - c\,)\, n \leq 2$
  - $n \leq 2/(7 - c)$
  - Pick $c = 7$ and $n_0 = 1$

Note

- The values of positive constants n_0 and c **are not unique** when proof the asymptotic formulas
- Example: show that $100n + 5 = O(n^2)$
  - $100n + 5 \leq 100n + n = 101n \leq 101n^2 \qquad \forall\ n \geq 5$

    $n_0 = 5$ and c = 101 are constants need to determine
  - $100n + 5 \leq 100n + 5n = 105n \leq 105n^2\ \forall\ n \geq 1$

    $n_0 = 1$ and $c = 105$ are also constants need to determine
- Only need to find **some** positive constants c and n_0 satisfying the equality in the definition of asymptotic notation

Big-Oh Examples

$O(g(n))$ = $\{f(n)$: $\exists$ positive constants c and n_0, such that $\forall n \geq n_0$, we have $0 \leq f(n) \leq cg(n)\}$

- Example 3: Show that $3n^3 + 20n^2 + 5$ is $O(n^3)$

  Need constants c and n_0 such that $3n^3 + 20n^2 + 5 \leq \boldsymbol{cn^3}$ for $n \geq n_0$

  this is true for $c = 4$ and $n_0 = 21$

- Example 4: Show that $3 \log n + 5$ is $O(\log n)$

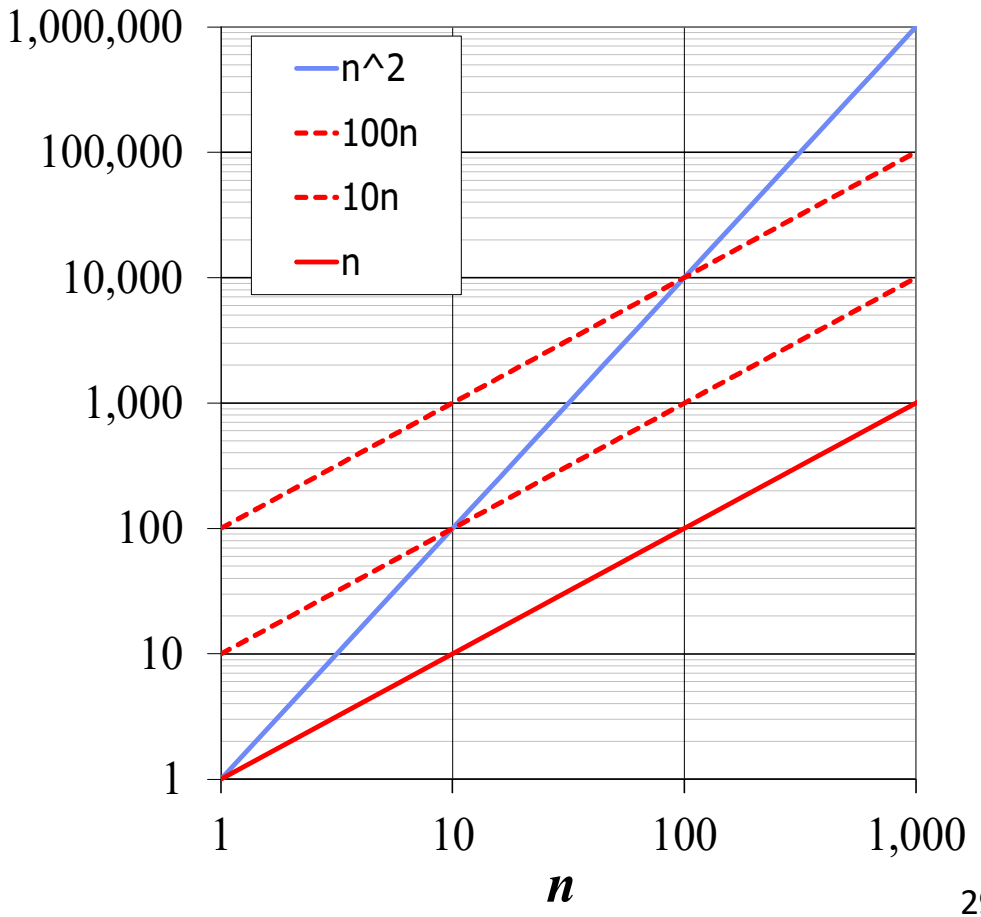
  Need constants c and n_0 such that $3 \log n + 5 \leq c \log n$ for $n \geq n_0$

  this is true for $c = 8$ and $n_0 = 2$

Big-Oh Examples

$O(g(n))$ = $\{f(n)$: ∃ positive constants c and n_0, such that $\forall n \geq n_0$, we have $0 \leq f(n) \leq cg(n)\}$

- Example 5: the function n^2 is not $O(n)$
  - $n^2 \leq cn$
  - $n \leq c$
  - The above inequality cannot be satisfied since c must be a constant



Big-Oh and Growth Rate

- The big-Oh notation gives an upper bound on the growth rate of a function
- The statement "$f(n)$ is $O(g(n))$" means that the growth rate of $f(n)$ is no more than the growth rate of $g(n)$
- We can use the big-Oh notation to rank functions according to their growth rate

	$f(n)$ is $O(g(n))$	$g(n)$ is $O(f(n))$
$g(n)$ grows more	Yes	No
$f(n)$ grows more	No	Yes
Same growth	Yes	Yes

Inappropriate Expressions

$$f(n) \leq O(g(n))$$

$$f(n) \geq O(g(n))$$

Big-Oh Examples

- $50n^3 + 20n + 4$ is $O(n^3)$
  - Would be correct to say is $O(n^3+n)$
    - Not useful, as n^3 exceeds by far n, for large values
  - Would be correct to say is $O(n^5)$
    - OK, but $g(n)$ should be as close as possible to $f(n)$
- $3\log(n) + \log(\log(n)) = O(\ ?\ )$

•Simple Rule: Drop lower order terms and constant factors

Useful Big-Oh Rules

- If $f(n)$ is a polynomial of degree d: $f(n) = a_0 + a_1 n + a_2 n^2 + \ldots + a_d n^d$

then $f(n)$ is $O(n^d)$, i.e.,

1. Drop lower-order terms
2. Drop constant factors

Example: $3n^3 + 20n^2 + 5$ is $O(n^3)$

- If $f(n) = O(n^k)$ then $f(n) = O(n^p)$ with $\forall\ p > k$

Example: $2n^2 = O(n^2)$ then $2n^2 = O(n^3)$

When evaluate asymptotic $f(n) = O(g(n))$, we want to find function $g(n)$ with a slower growth rate as possible

- Use the smallest possible class of functions

Example: Say "$2n$ is $O(n)$" instead of "$2n$ is $O(n^2)$"

- Use the simplest expression of the class

Example: Say "$3n + 5$ is $O(n)$" instead of "$3n + 5$ is $O(3n)$"

O Notation Examples

- All these expressions are O(n):
  - n, $3n$, $61n + 5$, $22n – 5$, …
- All these expressions are O(n^2):
  - n^2, $9\ n^2$, $18\ n^2 + 4n – 53$, …
- All these expressions are O($n \log n$):
  - $n(\log n)$, $5n(\log 99n)$, $18 + (4n – 2)(\log (5n + 3))$, …

Properties

- If $f(n)$ is $O(g(n))$ then $af(n)$ is $O(g(n))$ for any a
- If $f(n)$ is $O(g_1(n))$ and $h(n)$ is $O(g_2(n))$ then
  - $f(n)+h(n)$ is $O(g_1(n)+g_2(n))$
  - $f(n)h(n)$ is $O(g_1(n)\ g_2(n))$
- If $f(n)$ is $O(g(n))$ and $g(n)$ is $O(h(n))$ then $f(n)$ is $O(h(n))$
- If $p(n)$ is a polynomial in n then $\log p(n)$ is $O(\log(n))$
- If $p(n)$ is a polynomial of degree d, then $p(n)$ is $O(n^d)$
- $n^x = O(a^n)$, for any fixed $x > 0$ and $a > 1$
  - An algorithm of order n to a certain power is better than an algorithm of order a (> 1) to the power of n
- $\log n^x$ is $O(\log n)$, for $x > 0$ – how?
- $\log^x n$ is $O(n^y)$ for $x > 0$ and $y > 0$
  - An algorithm of order $\log n$ (to a certain power) is better than an algorithm of n raised to a power y.

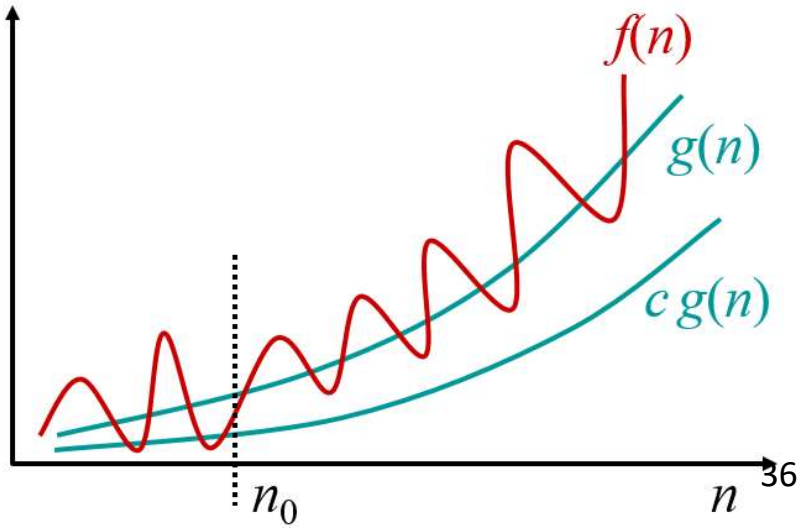
Ω-Omega notation

- For a given function $g(n)$, we denote by $\Omega(g(n))$ the set of functions

$$\Omega(g(n)) = \left\{ \begin{array}{c} f(n) : \text{there exist positive constants } c \text{ and } n_0 \text{ s.t.} \\ 0 \le cg(n) \le f(n) \text{ for all } n \ge n_0 \end{array} \right\}$$

Intuitively: Set of all functions whose *rate of growth* is the same as or higher than that of $g(n)$.

- We say: $g(n)$ is asymptotic lower bound of the function $f(n)$, to within a constant factor, and write $f(n) = \Omega(g(n))$.
- $f(n) = \Omega(g(n))$ means that there exists some constant c such that $f(n)$ is always $\ge cg(n)$ for large enough n.
- $\Omega(g(n))$ is the set of functions that go to infinity no slower than $g(n)$



$f(n) = \Theta(g(n)) \Rightarrow f(n) = \Omega(g(n))$
$\Theta(g(n)) \subset \Omega(g(n))$.

Omega Examples

$\Omega(g(n)) = \{f(n) : \exists$ positive constants c and n_0, such that $\forall n \geq n_0$, we have $0 \leq cg(n) \leq f(n)\}$

- Example 1: Show that $5n^2$ is $\Omega(n)$

  Need constants c and n_0 such that $\boldsymbol{cn} \leq 5n^2$ for $n \geq n_0$

  this is true for $c = 1$ and $n_0 = 1$

  Comment:

  - If $f(n) = \Omega(n^k)$ then $f(n) = \Omega(n^p)$ with $\forall\ p < k$.
  - When evaluate asymptotic $f(n) = \Omega(g(n))$, we want to find function $g(n)$ with a faster growth rate as possible

- Example 2: Show that $\sqrt{n} = \Omega(\lg n)$

Asymptotic notation in equations

Another way we use asymptotic notation is to simplify calculations:

- Use asymptotic notation in equations to replace expressions containing lower-order terms

Example:

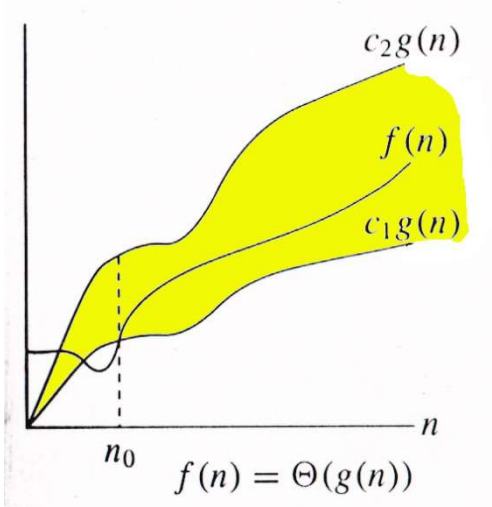
$$4n^3 + 3n^2 + 2n + 1 = 4n^3 + 3n^2 + \Theta(n)$$
$$= 4n^3 + \Theta(n^2) = \Theta(n^3)$$

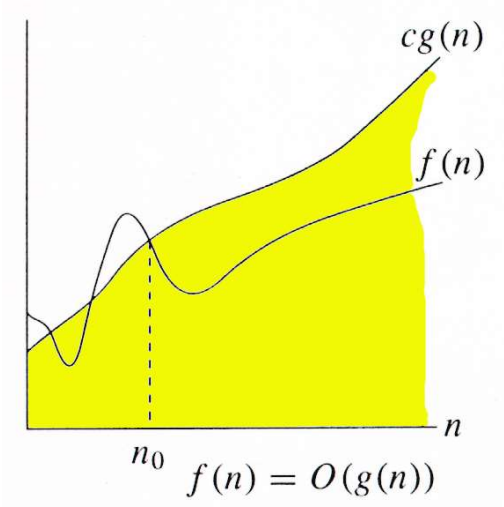
How to interpret?

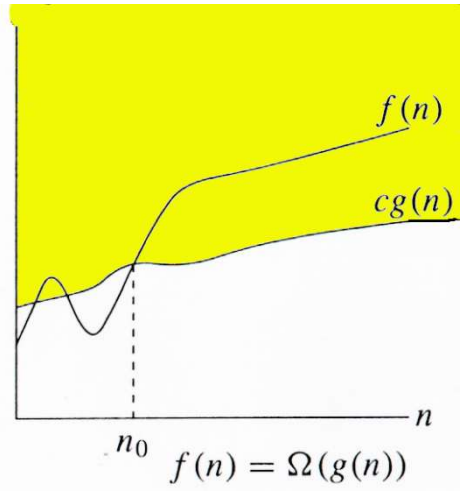
In equations, $\Theta(f(n))$ always stands for an ***anonymous function*** $g(n) \in \Theta(f(n))$

– In this example, we use $\Theta(n^2)$ stands for $3n^2 + 2n + 1$

Asymptotic notation







Graphic examples of $\boldsymbol{\Theta}$, O, and $\boldsymbol{\Omega}$

Theorem: For any two functions $f(n)$ and $g(n)$, we have $f(n) = \boldsymbol{\Theta}(g(n))$ if and only if

$$f(n) = O(g(n)) \text{ and } f(n) = \Omega(g(n))$$

Theorem: For any two functions $f(n)$ and $g(n)$, we have $f(n) = \Theta(g(n))$ if and only if $f(n) = O(g(n))$ and $f(n) = \Omega(g(n))$

Example 1: Show that $f(n) = 5n^2 = \Theta(n^2)$

Because:

- $5n^2 = O(n^2)$

  $f(n)$ is $O(g(n))$ if there is a constant $c > 0$ and an integer constant $n_0 \geq 1$ such that $f(n) \leq cg(n)$ for $n \geq n_0$

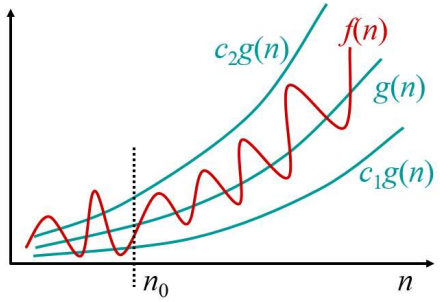
  let $c = 5$ and $n_0 = 1$

- $5n^2 = \Omega(n^2)$

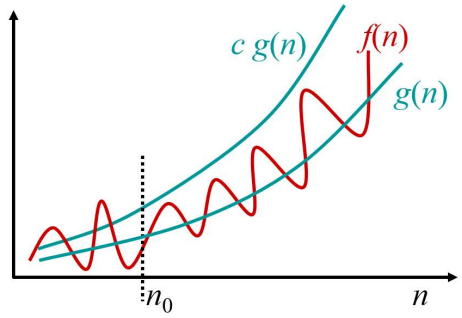
  $f(n)$ is $\Omega(g(n))$ if there is a constant $c > 0$ and an integer constant $n_0 \geq 1$ such that $f(n) \geq cg(n)$ for $n \geq n_0$

  let $c = 5$ and $n_0 = 1$

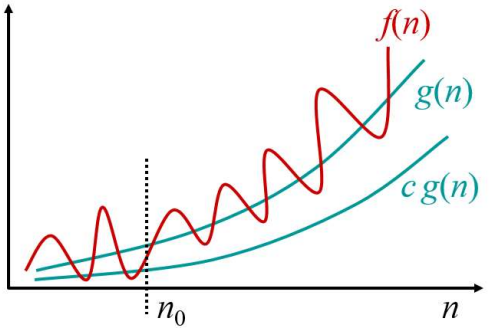
Therefore: $f(n) = \Theta(n^2)$



$f(n) = \Theta(g(n))$



$f(n) = O(g(n))$



$f(n) = \Omega(g(n))$

Theorem: For any two functions $f(n)$ and $g(n)$, we have $f(n) = \Theta(g(n))$ if and only if $f(n) = O(g(n))$ and $f(n) = \Omega(g(n))$

Example 2: Show that $f(n) = 3n^2 - 2n + 5 = \Theta(n^2)$

Because:

$3n^2 - 2n + 5 = O(n^2)$

$f(n)$ is $O(g(n))$ if there is a constant $c > 0$ and an integer constant $n_0 \geq 1$ such that $f(n) \leq cg(n)$ for $n \geq n_0$

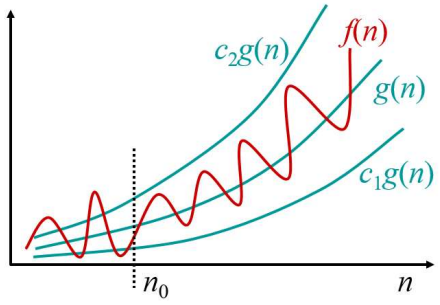
➔ pick c = ? and n_0 = ?

$3n^2 - 2n + 5 = \Omega(n^2)$

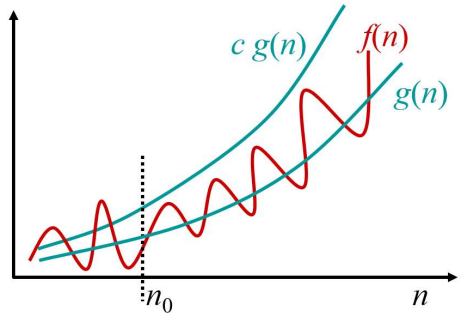
$f(n)$ is $\Omega(g(n))$ if there is a constant $c > 0$ and an integer constant $n_0 \geq 1$ such that $f(n) \geq cg(n)$ for $n \geq n_0$

➔ pick c = ? and n_0 = ?

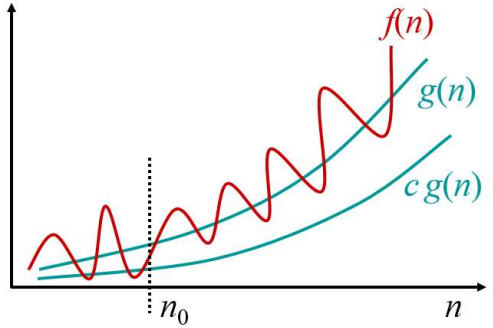
Therefore: $f(n) = \Theta(n^2)$



$f(n) = \Theta(g(n))$



$f(n) = O(g(n))$



$f(n) = \Omega(g(n))$

Exercise 1

Show that: $100n + 5 \neq \Omega(n^2)$

Ans: Contradiction

- Assume: $100n + 5 = \Omega(n^2)$

➔ $\exists\ c, n_0$ such that: $0 \leq cn^2 \leq 100n + 5$

- We have: $100n + 5 \leq 100n + 5n = 105n \quad \forall\ n \geq 1$
- Therefore: $cn^2 \leq 105n \Rightarrow n(cn - 105) \leq 0$
- As $n > 0 \Rightarrow cn - 105 \leq 0 \Rightarrow n \leq 105/c$

The above inequality cannot be satisfied since ***c*** must be a constant

Exercise 2

Show that: $n \neq \Theta(n^2)$

Ans: Contradiction

- Assume: $n = \Theta(n^2)$

➔ $\exists\ c_1, c_2, n_0$ such that: $c_1\ n^2 \leq n \leq c_2\ n^2\ \forall\ n \geq n_0$

➔ $n \leq 1/c_1$

The above inequality cannot be satisfied since c_1 must be a constant

Exercise 3:Show that

a) $6n^3 \neq \Theta(n^2)$

Ans: Contradiction

- Assume: $6n^3 = \Theta(n^2)$

➔ $\exists\ c_1, c_2, n_0$ such that: $c_1\ n^2 \leq 6n^3 \leq c_2\ n^2\ \forall\ n \geq n_0$

➔ $n \leq c_2/6\ \forall\ n \geq n_0$

The above inequality cannot be satisfied since c_2 must be a constant

b) $n \neq \Theta(\log_2 n)$

Ans: Contradiction

- Assume: $n = \Theta(\log_2 n)$

➔ $\exists\ c_1, c_2, n_0$ such that: $c_1\ \log_2 n \leq n \leq c_2\ \log_2 n\ \forall\ n \geq n_0$

➔ $n/\log_2 n \leq c_2\ \ \forall\ n \geq n_0$

The above inequality cannot be satisfied since c_2 must be a constant

Worst Case Analysis

- When to analyse an algorithm
  - Considering the cases only take
    - maximum amount of time

    (calculate the upper bound on running time of the algorithm)
  - If the algorithm is capable of solving cases in *f(n)*
    - then the worst case should not be greater than *c*f(n)*
- Useful if the algorithm is to be applied to cases
    - the upper bound of an algorithm must be known
- Example:
  - Response time for a nuclear power plant.

Average Time Analysis

- If the algorithm is going to be used many times
  - it is useful to know the average execution time on instances of size n
- It is harder to analyse the average case
  - the distribution of data should be known

Example: Insertion sorting average time is in the order of n^2

Best Case Analysis

- When to analyse an algorithm
  - Considering the cases only take
    - minimum amount of time

    (calculate the lower bound on running time of the algorithm)

The way to talk about the running time

- When people say "The running time for this algorithm is $O(f(n))$", it means that **the worst case running time is $O(f(n))$** (that is, no worse than $c*f(n)$ for large n, since big Oh notation gives an upper bound).
  - It means the worst case running time could be determined by some function $g(n) \in O(f(n))$
- When people say "The running time for this algorithm is $\Omega(f(n))$", it means that **the best case running time is $\Omega(f(n))$** (that is, no better than $c*f(n)$ for large n, since big Omega notation gives a lower bound).
  - It means the best case running time could be determined by some function $g(n) \in \Omega(f(n))$

Exercise

- Order the following functions by their asymptotic growth rates
  1. $n\log_2 n$
  2. $\log_2 n^3$
  3. n^2
  4. $n^{2/5}$
  5. $2^{log_2 n}$
  6. $\log_2(\log_2 n)$
  7. $\mathrm{Sqr}(\log_2 n)$

The way to remember these notations

Theta	$f(n) = \Theta(g(n))$	$f(n) \approx c\, g(n)$
Big Oh	$f(n) = O(g(n))$	$f(n) \leq c\, g(n)$
Big Omega	$f(n) = \Omega(g(n))$	$f(n) \geq c\, g(n)$
Little Oh	$f(n) = o(g(n))$	$f(n) < c\, g(n)$
Little Omega	$f(n) = \omega(g(n))$	$f(n) > c\, g(n)$

Properties

- Transitivity (truyền ứng)

$$f(n) = \Theta(g(n)) \;\&\; g(n) = \Theta(h(n)) \Rightarrow f(n) = \Theta(h(n))$$
$$f(n) = O(g(n)) \;\&\; g(n) = O(h(n)) \Rightarrow f(n) = O(h(n))$$
$$f(n) = \Omega(g(n)) \;\&\; g(n) = \Omega(h(n)) \Rightarrow f(n) = \Omega(h(n))$$

- Reflexivity

$$f(n) = \Theta(f(n)) \qquad f(n) = O(g(n)) \qquad f(n) = \Omega(g(n))$$

- Symmetry (đối xứng)

$$f(n) = \Theta(g(n)) \text{ if and only if } g(n) = \Theta(f(n))$$

- Transpose Symmetry (Đối xứng chuyển vị)

$$f(n) = O(g(n)) \text{ if and only if } g(n) = \Omega(f(n))$$

Example: $A = 5n^2 + 100n$, $B = 3n^2 + 2$. Show that $A \in \Theta(B)$

Ans: $A \in \Theta(n^2)$, $n^2 \in \Theta(B) \Rightarrow A \in \Theta(B)$

Limits

- $\lim_{n\to\infty} [f(n) / g(n)] = 0 \Rightarrow f(n) \in o(g(n))$
- $\lim_{n\to\infty} [f(n) / g(n)] < \infty \Rightarrow f(n) \in O(g(n))$
- $0 < \lim_{n\to\infty} [f(n) / g(n)] < \infty \Rightarrow f(n) \in \Theta(g(n))$
- $0 < \lim_{n\to\infty} [f(n) / g(n)] \Rightarrow f(n) \in \Omega(g(n))$
- $\lim_{n\to\infty} [f(n) / g(n)] = \infty \Rightarrow f(n) \in \omega(g(n))$
- $\lim_{n\to\infty} [f(n) / g(n)]$ undefined $\Rightarrow$ can't say

Exercise: Express functions in A in asymptotic notation using functions in B.

A	B	
$\log_3(n^2)$	**$\log_2(n^3)$**	**$A \in \Theta(B)$**

$\log_b a = \log_c a / \log_c b$; A = 2lg$n$ / lg3, B = 3lg$n$, A/B =2/(3lg3) $\Rightarrow$ A $\in \Theta$(B)

$n^{\lg 4}$	**$3^{\lg n}$**	**$A \in \omega(B)$**

$a^{\log b} = b^{\log a}$; B =$3^{\lg n}$=$n^{\lg 3}$; A/B =$n^{\lg(4/3)} \to \infty$ as $n\to\infty \Rightarrow$ A $\in \omega$ (B)

Exercise

Show that

1) $3n^2 - 100n + 6 = O(n^2)$
2) $3n^2 - 100n + 6 = O(n^3)$
3) $3n^2 - 100n + 6 \neq O(n)$
4) $3n^2 - 100n + 6 = \Omega(n^2)$
5) $3n^2 - 100n + 6 \neq \Omega(n^3)$
6) $3n^2 - 100n + 6 = \Omega(n)$
7) $3n^2 - 100n + 6 = \mathbf{\Theta}(n^2)$
8) $3n^2 - 100n + 6 \neq \mathbf{\Theta}(n^3)$
9) $3n^2 - 100n + 6 \neq \mathbf{\Theta}(n)$

Exercise

$3n^2 - 100n + 6 = O(n^2)$ since for $c = 3,\ 3n^2 > 3n^2 - 100n + 6$

$3n^2 - 100n + 6 = O(n^3)$ since for $c = 1,\ n^3 > 3n^2 - 100n + 6$ when $n > 3$

$3n^2 - 100n + 6 \neq O(n)$ since for any $c,\ cn < 3n^2$ when $n > c$

$3n^2 - 100n + 6 = \Omega(n^2)$ since for $c = 2,\ 2n^2 < 3n^2 - 100n + 6$ when $n > 100$

$3n^2 - 100n + 6 \neq \Omega(n^3)$ since for $c = 3,\ 3n^2 - 100n + 6 < n^3$ when $n > 3$

$3n^2 - 100n + 6 = \Omega(n)$ since for any $c,\ cn < 3n^2 - 100n + 6$ when $n > 100$

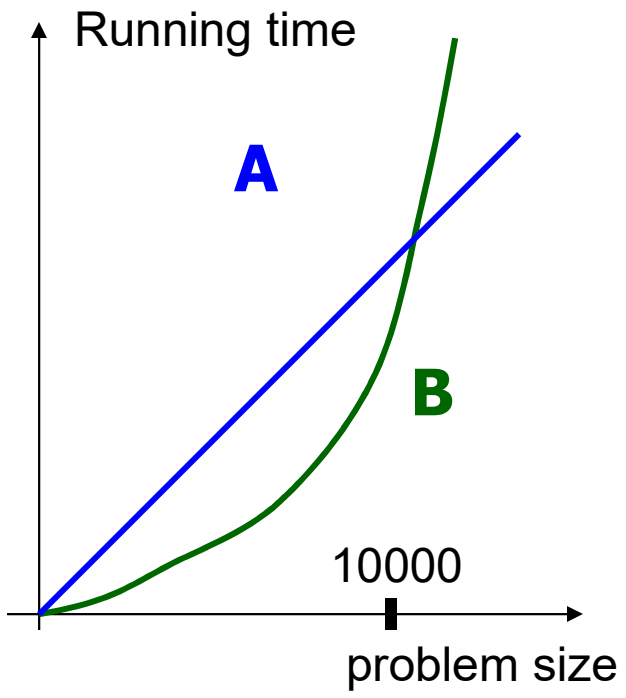
$3n^2 - 100n + 6 = \Theta(n^2)$ since both O and Ω apply.

$3n^2 - 100n + 6 \neq \Theta(n^3)$ since only O applies.

$3n^2 - 100n + 6 \neq \Theta(n)$ since only Ω applies.

Final notes

- Even though in this course we focus on the asymptotic growth using big-Oh notation, practitioners do care about constant factors occasionally
- Suppose we have 2 algorithms
  - Algorithm A has running time $30000n$
  - Algorithm B has running time $3n^2$
- Asymptotically, algorithm A is better than algorithm B
- However, if the problem size you deal with is always less than 10000, then the quadratic one is faster



Algorithm analysis example

Example 1. Consider sequential search algorithm to solve the problem:

- **Input:** *key*, *n* and sequence $s_1, s_2, \ldots, s_n$.
- **Output:** position in the sequence that *key* is located or n+1 if *key* is not in the sequence.

```
int  Linear_Search(s,n,key)
{
    i=0;
    do
          i=i+1;
    while  (i<=n) or (key != s_i);
    return i;
}
```

NGUYỄN KHÁNH PHƯƠNG
Bộ môn KHMT – ĐHBK HN

Algorithm analysis example

```
int  Linear_Search(s,n,key)
{
     i=0;
     do
          i=i+1;
     while  (i<=n) or (key != s_i);
     return i;
}
```

Need to evaluate the **best, worst, average** computation time of the algorithm with an input length of n.

Apparently, the computation time of the algorithm can be evaluated by the number of executions of the statement `i = i + 1` in the do-while loop.

Best-case running time: If $s_1 = key$, the statement `i = i + 1` in the body of the do-while loop executes once. Therefore, the best-case running time of the algorithm is $\Theta(1)$.

Worst-case running time: If the *key* is not presented in the given sequence, the statement `i = i + 1` executes n times. So the worst-case running time of the algorithm is $\Theta(n)$.

Algorithm analysis example

```
int  Linear_Search(s,n,key)
{
    i=0;
    do
        i=i+1;
    while  (i<=n) or (key != s_i);
    return i;
}
```

- **Average-case running time:**
  - If *key* is found at position *i* of sequence ($key = s_i$) then the statement $i = i+1$ is executed *i* times ($i = 1, 2, ..., n$),
  - If *key* is not found in the sequence then the statement $i = i+1$ is executed *n* times.

→ Thus, the average number of times that the statement $i = i+1$ being executed is

$$[(1 + 2 + \ldots + n) + n] /(n+1) = [n(n+1)/2 + n]/(n+1).$$

We have:

$$n/4 \leq [n(n+1)/2 + n]/(n+1) \leq n \text{ với mọi } n \geq 1.$$

Thus, the average-case running time of the algorithm is $\Theta(n)$.

Algorithm analysis example

Example 2: Using big-Oh asymptotic to evaluate the running time $T(n)$ for the following source codes:

```
a)   for (int i = 1; i<=n; i++)
         for (int j = 1; j<= i*i*i; j++)
          for (int k = 1; k<=n; k++)
                    x = x + 1;
```

- Answer:

$$T(n) = \sum_{i=1}^{n} \sum_{j=1}^{i^3} \sum_{k=1}^{n} 1$$

$$= \sum_{i=1}^{n} \sum_{j=1}^{i^3} n = \sum_{i=1}^{n} n(\sum_{j=1}^{i^3} 1) = \sum_{i=1}^{n} ni^3$$

$$= n\sum_{i=1}^{n} i^3 \le n\sum_{i=1}^{n} n^3 = n^4 \sum_{i=1}^{n} 1 = n^5$$

Thus, $T(n) = O(n^5)$

Algorithm analysis example

Example 2: Using big-Oh asymptotic to evaluate the running time $T(n)$ for the following source codes:

b)
```
int x = 0;
    for (int i = 1; i <= n; i *= 2)
        x=x+1;
```

- Answer:

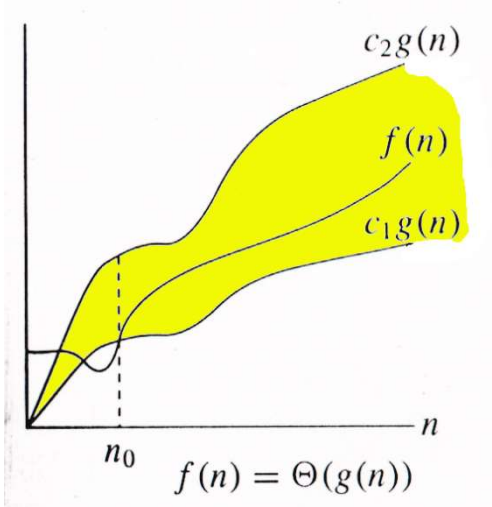
The loop **for** is executed $\log_2 n$ times, thus $T(n) = O(\log_2 n)$.

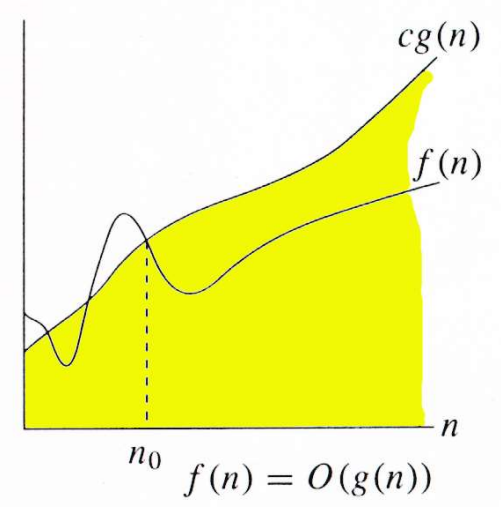
c)
```
int x = 0;
    for (int i = n; i > 0; i /= 2)
        x=x+1;
```

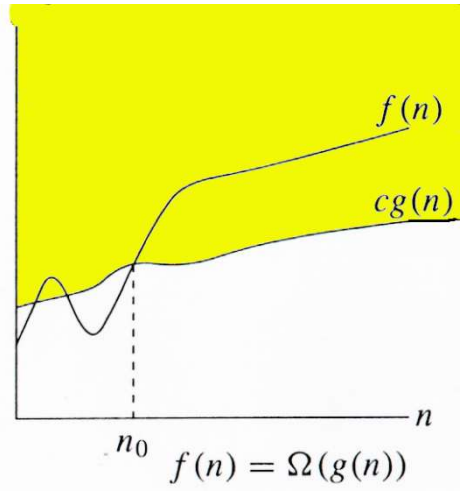
- Answer:

The loop **for** is executed …..

Exercise







Graphs illustrate the asymptotic notations **Θ**, *O*, and **Ω**

- $f(n)$ and $g(n)$ have the same rath of growth — $f(n) = \Theta(g(n))$
- $f(n)$ has the rate of growth not greater than that of $g(n)$ — $f(n) = O(g(n))$
- $f(n)$ has the rate of growth not smaller than that of $g(n)$ — $f(n) = \Omega(g(n))$

1) $3n^2 - 100n + 6$? $O(n)$
2) $3n^2 - 100n + 6$? $O(n^2)$
3) $3n^2 - 100n + 6$? $O(n^3)$
4) $3n^2 - 100n + 6$? $\Omega(n)$
5) $3n^2 - 100n + 6$? $\Omega(n^2)$
6) $3n^2 - 100n + 6$? $\Omega(n^3)$
7) $3n^2 - 100n + 6$? $\mathbf{\Theta}(n)$
8) $3n^2 - 100n + 6$? $\mathbf{\Theta}(n^2)$
9) $3n^2 - 100n + 6$? $\mathbf{\Theta}(n^3)$

Master Theorem

- Let T(n) be a monotonically increasing function that satisfies

$$T(n) = a\,T(n/b) + f(n)$$

$$T(1) = c$$

where $a \geq 1$, $b \geq 2$, $c > 0$. If $f(n)$ is $\Theta(n^d)$ where $d \geq 0$ then

$$T(n) = \begin{cases} \Theta(n^d) & \text{if } a < b^d \\ \Theta(n^d \log n) & \text{If } a = b^d \\ \Theta(n^{\log_b a}) & \text{if } a > b^d \end{cases}$$

Master Theorem: Pitfalls

- Let T(n) be a monotonically increasing function that satisfies

  $T(n) = a\,T(n/b) + f(n)$

  $T(1) = c$

  where $a \geq 1$, $b \geq 2$, $c>0$. If $f(n)$ is $\Theta(n^d)$ where $d \geq 0$ then

$$T(n) = \begin{cases} \Theta(n^d) & \text{if } a < b^d \\ \Theta(n^d \log n) & \text{If } a = b^d \\ \Theta(n^{\log_b a}) & \text{if } a > b^d \end{cases}$$

- You cannot use the Master Theorem if
  - T(n) is not monotone, e.g. T(n) = sin(n)
  - f(n) is not a polynomial, e.g. T(n) = 2T(n/2)+ 2^n
  - b cannot be expressed as a constant, e.g. $T(n) = T(\sqrt{n})$

Master Theorem: Example 1

- Let $T(n) = T(n/2) + \frac{1}{2} n^2 + n$. What are the parameters?

  a = 1

  b = 2

  d = 2

  Therefore, which condition applies?

  $1 < 2^2$, case 1 applies

- We conclude that $T(n) \in \Theta(n^d) = \Theta(n^2)$

Master Theorem

- Let $T(n)$ be a monotonically increasing function that satisfies

  $T(n) = a\,T(n/b) + f(n)$

  $T(1) = c$

  where $a \geq 1$, $b \geq 2$, $c > 0$. If $f(n)$ is $\Theta(n^d)$ where $d \geq 0$ then

$$T(n) = \begin{cases} \Theta(n^d) & \text{if } a < b^d \\ \Theta(n^d \log n) & \text{If } a = b^d \\ \Theta(n^{\log_b a}) & \text{if } a > b^d \end{cases}$$

Master Theorem: Example 2

- Let $T(n) = 2\,T(n/4) + \sqrt{n} + 42$. What are the parameters?

  a = 2

  b = 4

  d = 1/2

  Therefore, which condition applies?

  $2 = 4^{1/2}$, case 2 applies

- We conclude that $T(n) \in \Theta(n^d \log n) = \Theta(\log n \sqrt{n})$

Master Theorem

- Let $T(n)$ be a monotonically increasing function that satisfies

  $T(n) = a\,T(n/b) + f(n)$

  $T(1) = c$

  where $a \geq 1$, $b \geq 2$, $c > 0$. If $f(n)$ is $\Theta(n^d)$ where $d \geq 0$ then

$$T(n) = \begin{cases} \Theta(n^d) & \text{if } a < b^d \\ \Theta(n^d \log n) & \text{If } a = b^d \\ \Theta(n^{\log_b a}) & \text{if } a > b^d \end{cases}$$

Master Theorem: Example 3

- Let T(n)= 3 T(n/2) + (3/4)n + 1. What are the parameters?

  a = 3

  b = 2

  d = 1

  Therefore, which condition applies?

  $3 > 2^1$, case 3 applies

- We conclude that $T(n) \in \Theta(n^{\log_b a}) = \Theta(n^{\log_2 3})$

Master Theorem

- Let T(n) be a monotonically increasing function that satisfies

  $T(n) = a\,T(n/b) + f(n)$

  $T(1) = c$

  where $a \geq 1$, $b \geq 2$, $c>0$. If $f(n)$ is $\Theta(n^d)$ where $d \geq 0$ then

$$T(n) = \begin{cases} \Theta(n^d) & \text{if } a < b^d \\ \Theta(n^d \log n) & \text{If } a = b^d \\ \Theta(n^{\log_b a}) & \text{if } a > b^d \end{cases}$$

Master Theorem: 'Fourth' Condition

- Recall that we cannot use the Master Theorem if f(n) (the non-recursive cost) is not a polynomial.
- There is a limited 4th condition of the Master Theorem that allows us to consider polylogarithmic functions:

Corollary

If $f(n) \in \Theta(n^{\log_b a} \log^k n)$ for some $k \geq 0$ then

$$T(n) \in \Theta(n^{\log_b a} \log^{k+1} n)$$

Master Theorem

- Let T(n) be a monotonically increasing function that satisfies

  $T(n) = a\ T(n/b) + f(n)$

  $T(1) = c$

  where $a \geq 1$, $b \geq 2$, $c > 0$. If $f(n)$ is $\Theta(n^d)$ where $d \geq 0$ then

$$T(n) = \begin{cases} \Theta(n^d) & \text{if } a < b^d \\ \Theta(n^d \log n) & \text{If } a = b^d \\ \Theta(n^{\log_b a}) & \text{if } a > b^d \end{cases}$$

‘Fourth’ Condition: Example

- Say we have the following recurrence relation

  $T(n) = 2\,T(n/2) + n\log n$

- Clearly, $a=2$, $b=2$, but $f(n)$ is not a polynomial.
- However, we have $f(n) \in \Theta(n\log n)$, $k=1$
- Therefore, by the 4th condition of the Master Theorem we can say that

$$T(n) \in \Theta(n^{\log_b a} \log^{k+1} n) = \Theta(n^{\log_2 2} \log^2 n) = \Theta(n \log^2 n)$$

Corollary

If $f(n) \in \Theta(n^{\log_b a} \log^k n)$ *for some* $k \geq 0$ *then*

$$T(n) \in \Theta(n^{\log_b a} \log^{k+1} n)$$

CONTENTS

NGUYỄN KHÁNH PHƯƠNG
Bộ môn KHMT – ĐHBK HN

3. Generating algorithm

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
Bộ môn KHMT – ĐHBK HN

3.1. 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

NGUYỄN KHÁNH PHƯƠNG
Bộ môn KHMT – ĐHBK HN

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.

3. Generating algorithm

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
Bộ môn KHMT – ĐHBK HN

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
Bộ môn KHMT – ĐHBK HN

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
Bộ môn KHMT – ĐHBK HN

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.

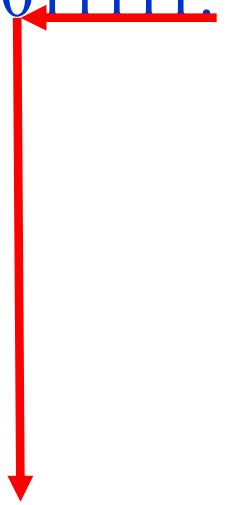
Generate binary strings of length *n*

- Find the first i (in the order $i = n, n\text{-}1, ..., 1$) 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
```

NGUYỄN KHÁNH PHƯƠNG
Bộ môn KHMT – ĐHBK HN

Successive_Generation algorithm

```
def  Next_Bit_String( ):
#Generate the next binary string in the dictionary order
#of the current string  b1 b2 ... bn ≠ 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 $i = n, n\text{-}1, ..., 1$) 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.

3. Generating algorithm

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
Bộ môn KHMT – ĐHBK HN

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

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

Solution:

- Lexicographic order:

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

$$a = (a_1, a_2, \dots, a_m)$$

satisfying

$$1 \le a_1 < a_2 < \dots < a_m \le n.$$

NGUYỄN KHÁNH PHƯƠNG
Bộ môn KHMT – ĐHBK HN

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:

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

$$a_k < a'_k .$$

NGUYỄN KHÁNH PHƯƠNG
Bộ môn KHMT – ĐHBK HN

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
Bộ môn KHMT – ĐHBK HN

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

Successive_Generation algorithm:

- The first subset is (1, 2, ... , m)
- The last subset is (n-m+1, n-m+2, ..., n).
- Assume $a=(a_1, a_2, \dots, a_m)$ 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 $a_1, a_2, \dots, a_m$: find 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 $j = i+1, i+2, \dots, m$

NGUYỄN KHÁNH PHƯƠNG
Bộ môn KHMT – ĐHBK HN

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 $a_1, a_2, \ldots, a_m$: find the first element $a_i \neq n\text{-}m\text{+}i$
- Replace a_i by $a_i + 1$
- Replace a_j by $a_i + j - i$, where $j = i+1, i+2, \ldots, m$

- We have i=2:

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

Value $n\text{-}m\text{+}i$: (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 (a1, a2,..., am)  ≠  (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 $a_1, a_2, \ldots, a_m$ the first element $a_i \neq n\text{-}m\text{+}i$
- Replace a_i by $a_i + 1$
- Replace a_j by $a_i + j - i$, where $j = i+1, i+2, \ldots, m$

3. Generating algorithm

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
Bộ môn KHMT – ĐHBK HN

Generate permutations of *n* elements

Problem: Give $X = \{1, 2, \ldots, n\}$, 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:

$$a = (a_1, a_2, \ldots, a_n)$$

satisfy

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

NGUYỄN KHÁNH PHƯƠNG
Bộ môn KHMT – ĐHBK HN

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 $(1 \le k \le n)$ such that:

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

$$a_k < a'_k .$$

NGUYỄN KHÁNH PHƯƠNG
Bộ môn KHMT – ĐHBK HN

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

NGUYỄN KHÁNH PHƯƠNG
Bộ môn KHMT – ĐHBK HN

Generate permutations of *n* elements

Successive_Generation algorithm:

- The first permutation: $(1, 2, \ldots, n)$
- The last permutation: $(n, n\text{-}1, \ldots, 1)$.
- Assume $a = (a_1, a_2, \ldots, a_n)$ is not the last permutation, then its next permutation could be built using the following rules:
  - Find from the left to right: 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 .

NGUYỄN KHÁNH PHƯƠNG
Bộ môn KHMT – ĐHBK HN

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 left to right: 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 (a1, a2, ..., an) ≠(n, n-1, ..., 1)   */
    // Find j being max index satisfying aj < aj+1 :
    j=n-1;  while (aj > aj+1 ) j=j-1;
    // Find ak being smallest number among those on the right aj and greater
    //than aj :
    k=n;  while (aj > ak ) k=k-1;
    Swap(aj, ak); //swap aj and ak
    // Invert the sequence aj+1 to an :
    r=n; s=j+1;
    while ( r>s )
    {
        Swap(ar, as);  // swap ar and as
        r=r-1; s= s+1;
    }
}
```

Successive_Generation algorithm

```
def Next_Permutation ( ):
```

#Generate next permutation $(a_1, a_2, ..., a_n) \neq (n, n\text{-}1, ..., 1)$

#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
    s = j+1
    while ( r > s ):
        Swap(a[r], a[s])
        r = r-1
        s = s+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

NGUYỄN KHÁNH PHƯƠNG
Bộ môn KHMT – ĐHBK HN

Backtracking (Thuật toán quay lui)

4.1. Algorithm diagram

4.2. 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

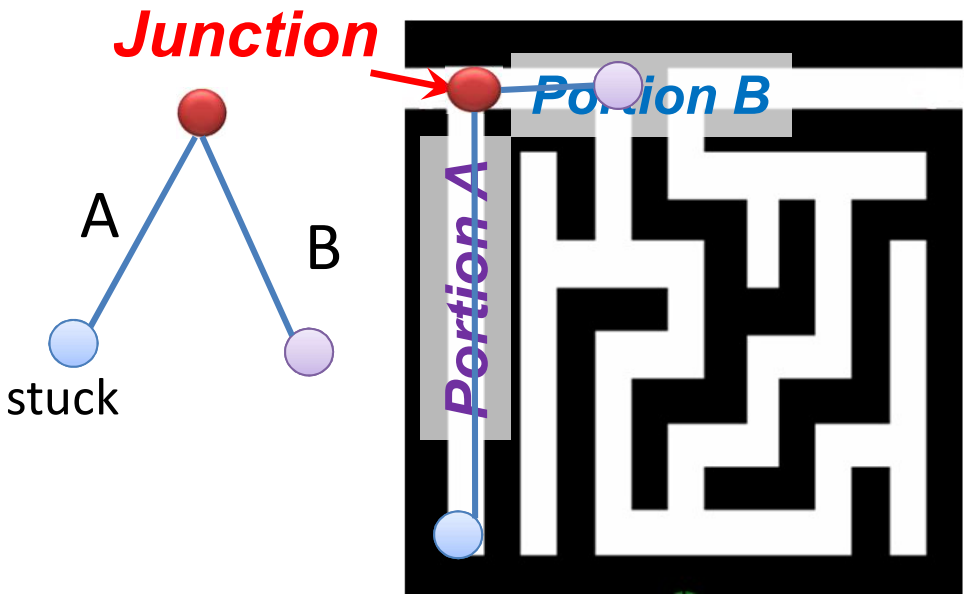
NGUYỄN KHÁNH PHƯƠNG
Bộ môn KHMT – ĐHBK HN

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**

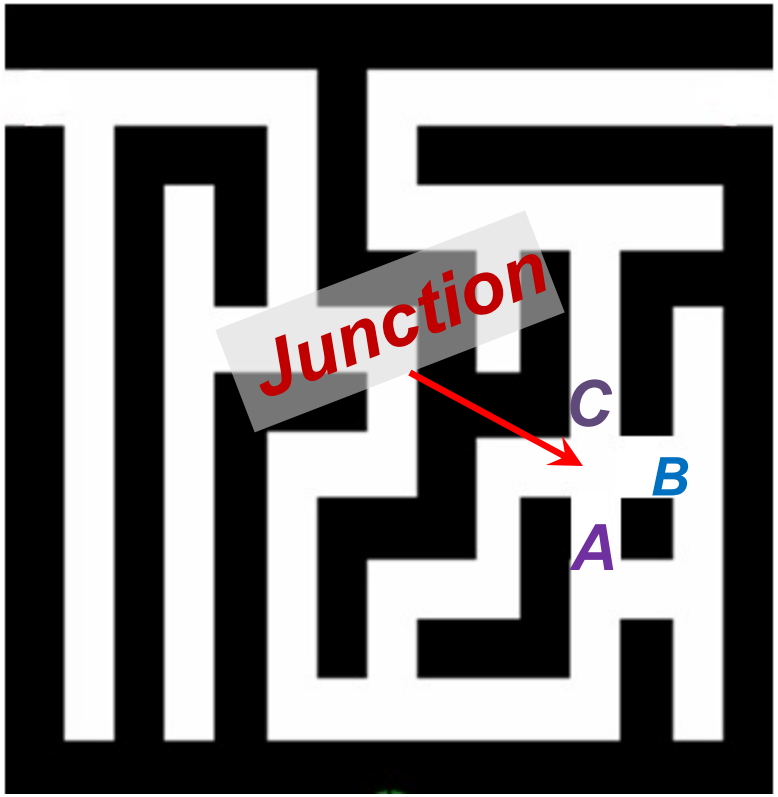


Backtracking idea

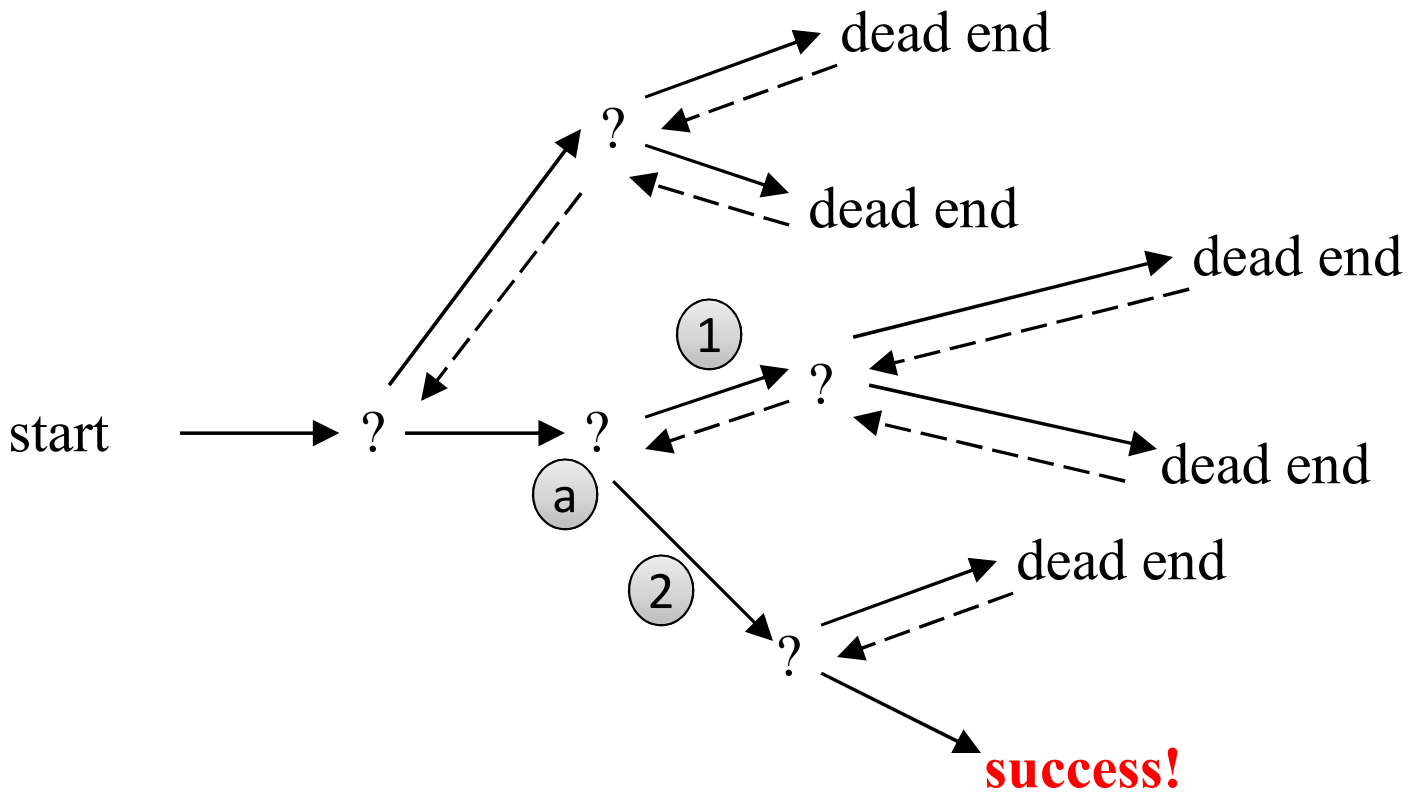
- Clearly, at a single junction you could have even more than 2 choices.

- 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) a

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. a

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,..., A_n *be finite sets. Denote*

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

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:

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

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

NGUYỄN KHÁNH PHƯƠNG
Bộ môn KHMT – ĐHBK HN

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:

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

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

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

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

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

Partial solution (Lời giải bộ phận)

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

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

$$(a_1, a_2, ..., a_k),$$

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

- 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,..., A_n be finite sets. Denote*

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

  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:

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

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:

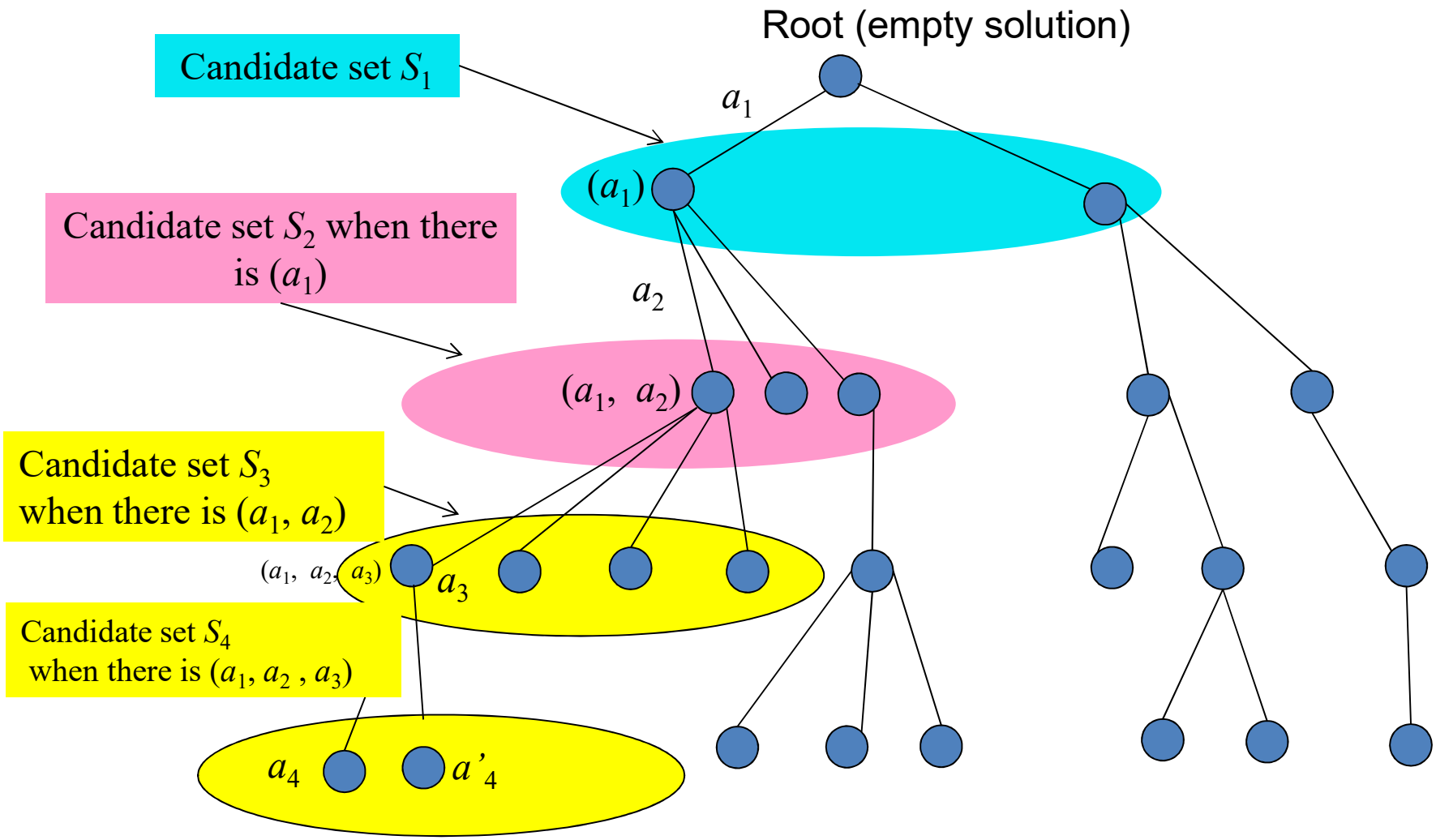
$$(a_1, a_2, ..., a_{k-1}, \mathbf{a_k})$$

  - 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:
    - $S_k \neq \varnothing$
    - $S_k = \varnothing$

Backtracking diagram

- ***$S_k \neq \varnothing$:*** 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.
- ***$S_k = \varnothing$:*** 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 1^{st} position, then the algorithm is finished.

Decision tree for backtracking



Backtracking algorithm (recursive)

```
void Try(int k)
{
   <Build Sk as the set to consist of candidates for the kth
      component of solution>;
  for y ∈ Sk  //Each candidate y of Sk
  {
     ak = y;
     if (k == n) then <Record (a1, a2, ..., ak) 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 Backtraking ( )
{
 k=1;
 <Build Sk>;
 while (k > 0)  {
    while (Sk ≠ ∅ ) {
        ak ← Sk; // Take ak from Sk
        if  <(k == n) >  then <Record (a1,a2,...,ak) as a
                                  complete solution >;
        else {
              k = k+1;
              <Build Sk>;
        }
    }
     k = k - 1;   // Backtracking
  }
}
```

The call to execute backtracking algorithm: Bactraking ( );

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` $y \in S_k$).
    - The efficiency of the enumeration algorithm depends on whether we can accurately identify these candidate sets.

NGUYỄN KHÁNH PHƯƠNG
Bộ môn KHMT – ĐHBK HN

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)

4.1. Algorithm diagram

4.2. 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
Bộ môn KHMT – ĐHBK HN

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:

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

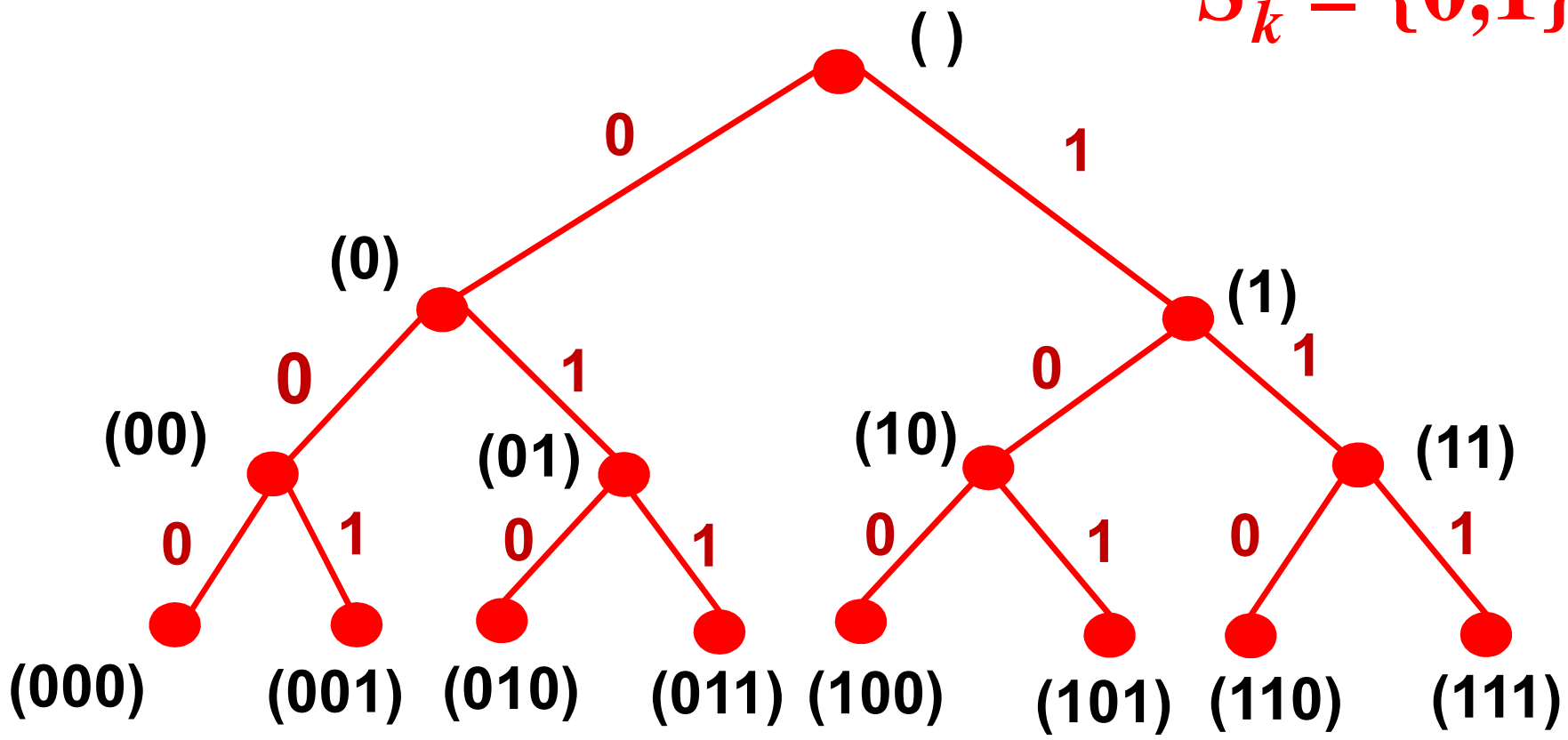
- We consider how to solve two issue keys to implement backtracking algorithm:
  - Build candidate set S_k: We have $S_1 = \{0, 1\}$. Assume we have binary string of length k-1 $(a_1, ..., a_{k-1})$, then $S_k = \{0,1\}$. Thus, the candidate sets for each position of the solution are determined.
  -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

NGUYỄN KHÁNH PHƯƠNG
Bộ môn KHMT – ĐHBK HN

Decision tree to enumerate binary strings of length 3

$S_k = \{0,1\}$



NGUYỄN KHÁNH PHƯƠNG
Bộ môn KHMT – ĐHBK HN

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;
}
```

NGUYỄN KHÁNH PHƯƠNG
Bộ môn KHMT – ĐHBK HN

Program in Python (Recursive)

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

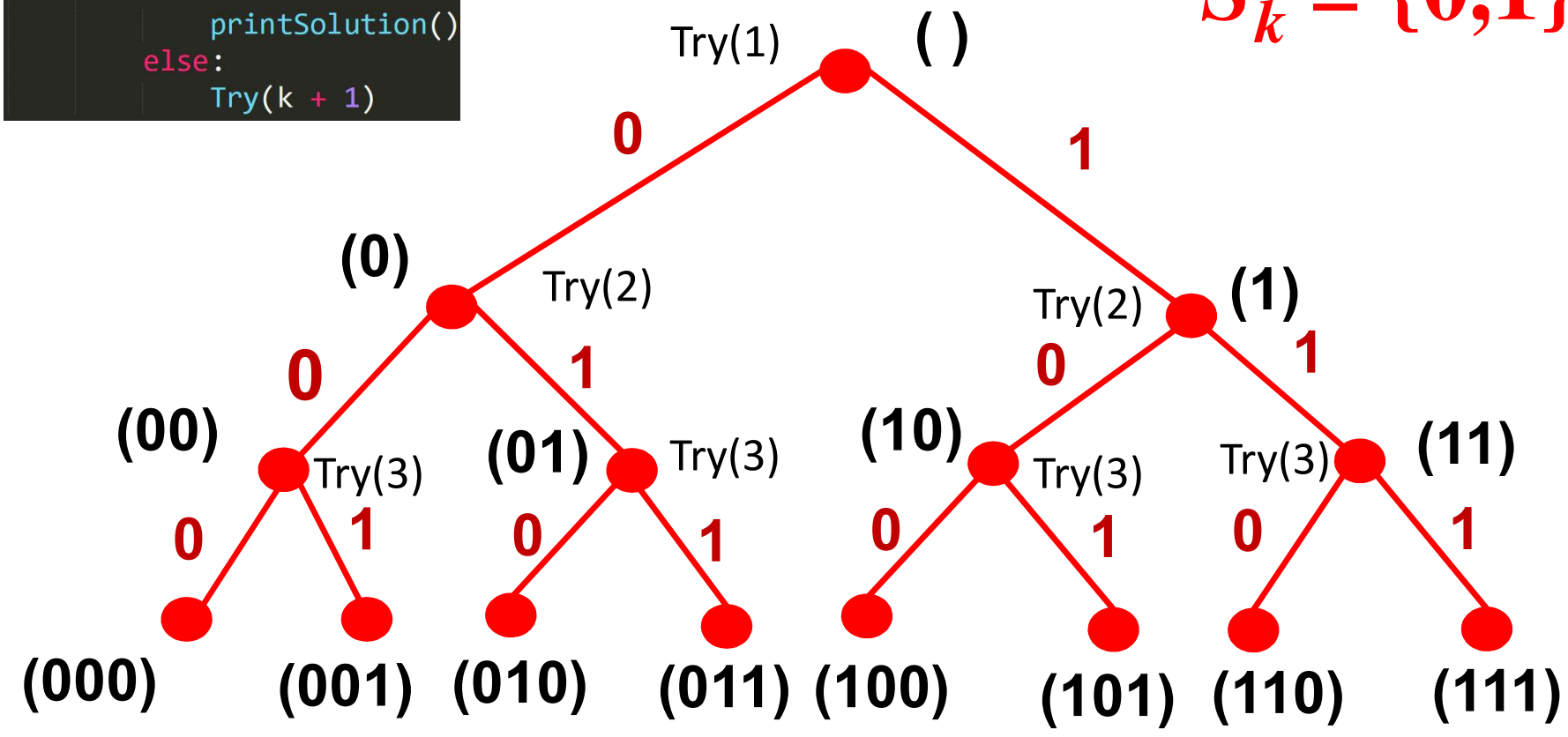
```python
def Try(k):
    for j in range(0, 2):
        a[k] = j
        if k == n:
            printSolution()
        else:
            Try(k + 1)
```

```python
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)
```

$$S_k = \{0,1\}$$



NGUYỄN KHÁNH PHƯƠNG
Bộ môn KHMT – ĐHBK HN

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
Bộ môn KHMT – ĐHBK HN

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)

4.1. Algorithm diagram

4.2. 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
Bộ môn KHMT – ĐHBK HN

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:

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

NGUYỄN KHÁNH PHƯƠNG
Bộ môn KHMT – ĐHBK HN

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: : $1 \le a_1 < a_2 < ... < a_m \le n$

  we have $S_1 = \{1, 2, ..., n-(m-1)\}$.
  - Assume the current subset is $(a_1, ..., a_{k-1})$, with the condition $a_{k-1} < a_k < ... < a_m \le n$, 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
```

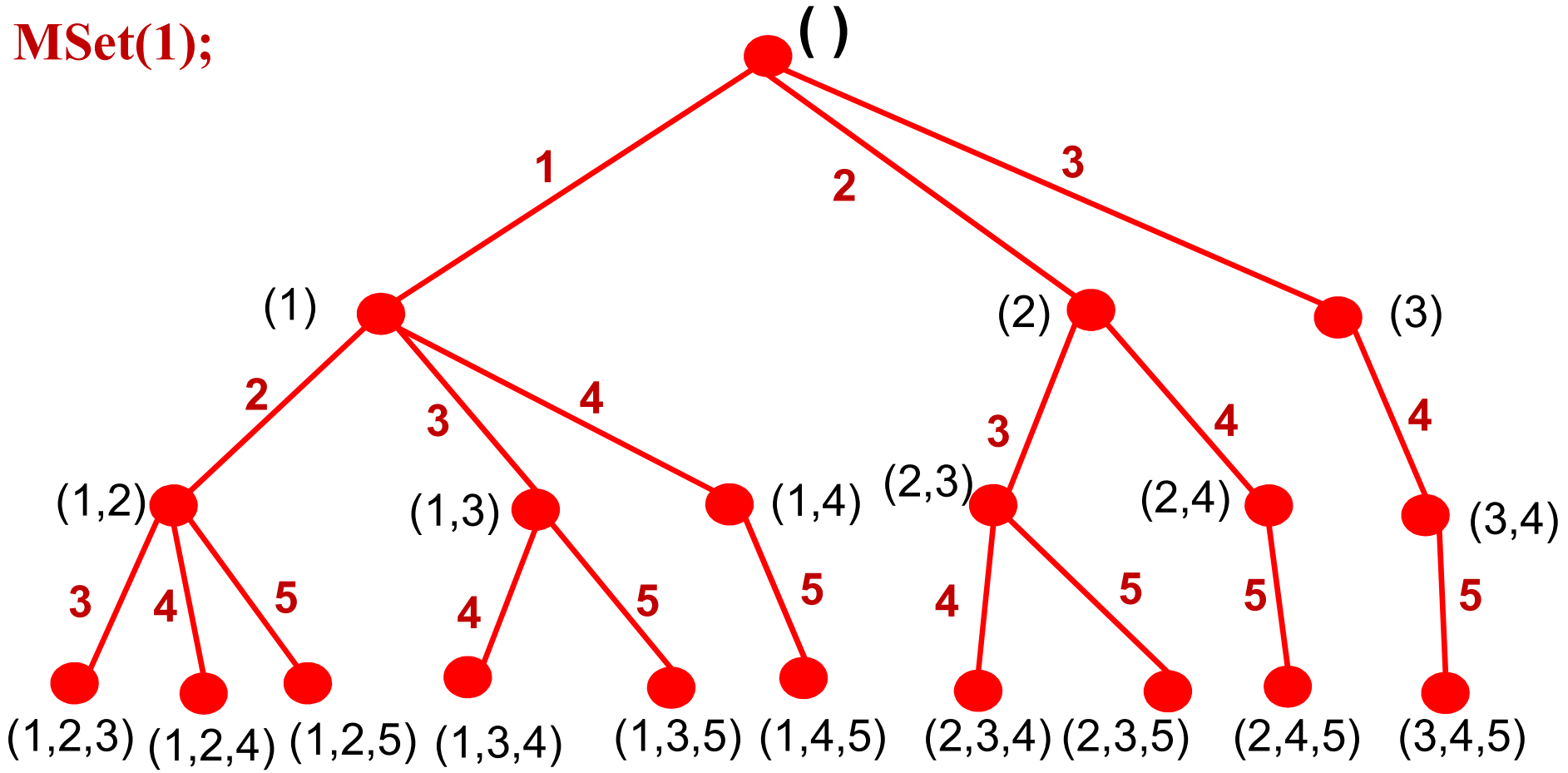
NGUYỄN KHÁNH PHƯƠNG
Bộ môn KHMT – ĐHBK HN

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)
```

NGUYỄN KHÁNH PHƯƠNG
Bộ môn KHMT – ĐHBK HN

Decision tree S(5,3)



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

Backtracking (Thuật toán quay lui)

4.1. Algorithm diagram

4.2. 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
Bộ môn KHMT – ĐHBK HN

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 Π_n

NGUYỄN KHÁNH PHƯƠNG
Bộ môn KHMT – ĐHBK HN

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
Bộ môn KHMT – ĐHBK HN

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:

```
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 k^th
     . . .
  }
```

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)

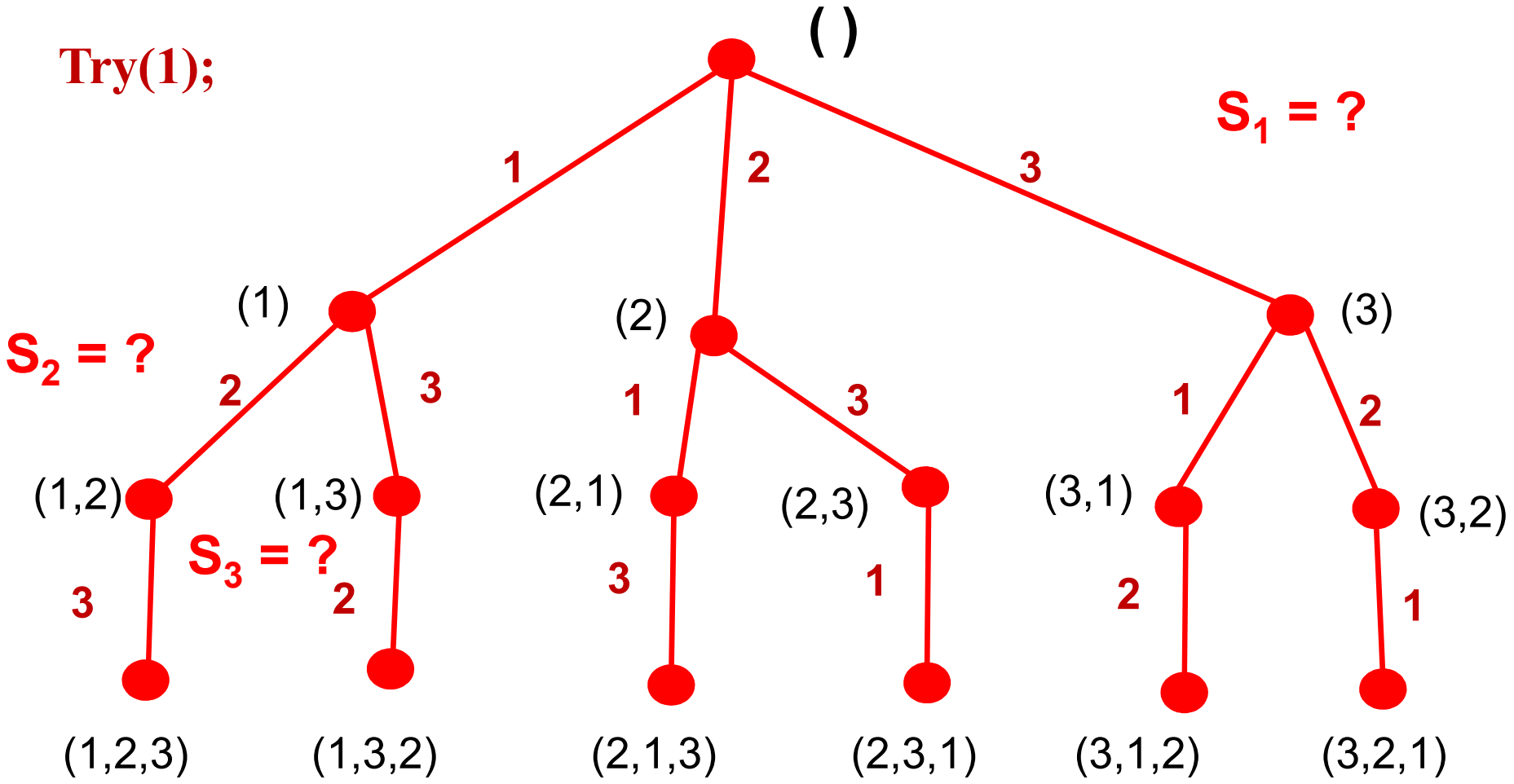
```
def printSolution():
    global count
    count = count + 1
    for i in range(1, n + 1):
        print(a[i], end=" ")
    print()

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



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

Exercise

k and *n* are non negative integers. The equation below has how many non negative-integer roots ?

$$x_1 + x_2 + x_3 + \cdots + x_k = n$$
$$x_1, x_2, \cdots, x_k \geq 0$$

The number of non-negative integer roots is:

C(*n*+*k*-1, *n*) = C(*n*+*k*-1, *k*-1) = (*n*+*k*-1)! / *n*!(*k*-1)!

k and *n* are non-negative integers. The equation below has how many positive-integer roots?

$$x_1 + x_2 + x_3 + \cdots + x_k = n$$
$$x_1, x_2, \cdots, x_k > 0$$

Set $y_i = x_i - 1 \geq 0$ ➔ $y_1 + y_2 + y_3 + \cdots + y_k = n - k$

$$y_1, y_2, \cdots, y_k \geq 0$$

The number of positive-integer roots is: (*n*-1)!/(*n*-*k*)!(*k*-1)!

Exercise

Enumerate all positive-integer roots of equation:

$$x_1 + x_2 + x_3 + \cdots + x_k = n$$

$$x_1, x_2, \cdots, x_k > 0$$

Backtracking: Find value for each variable x_i of the equation: at iteration i^{th} (i=1,..,n): we need to find value of variable x_i

- Assume we already have partial solution (x_1, x_2, …, x_{i-1}): it means already know values of (i-1)th first variables x_1, x_2, …, x_{i-1}
- Now we build the i^{th} component of solution: it means we determine value for variable x_i. We do as following:
  - Calculate the sum of (i-1) first variables that already know values: M = x_1 + x_2+ …+x_{i-1}
  - Sum of (k-i) variables x_{i+1}, … ,x_k has the least value = $k - i$
  - Maximum value of x_i could be: U = n – M – (k – i)

  ➔ variable x_i can only get the value in range: $1 \le x_i \le$ U

Backtracking: Function to recognize candidate

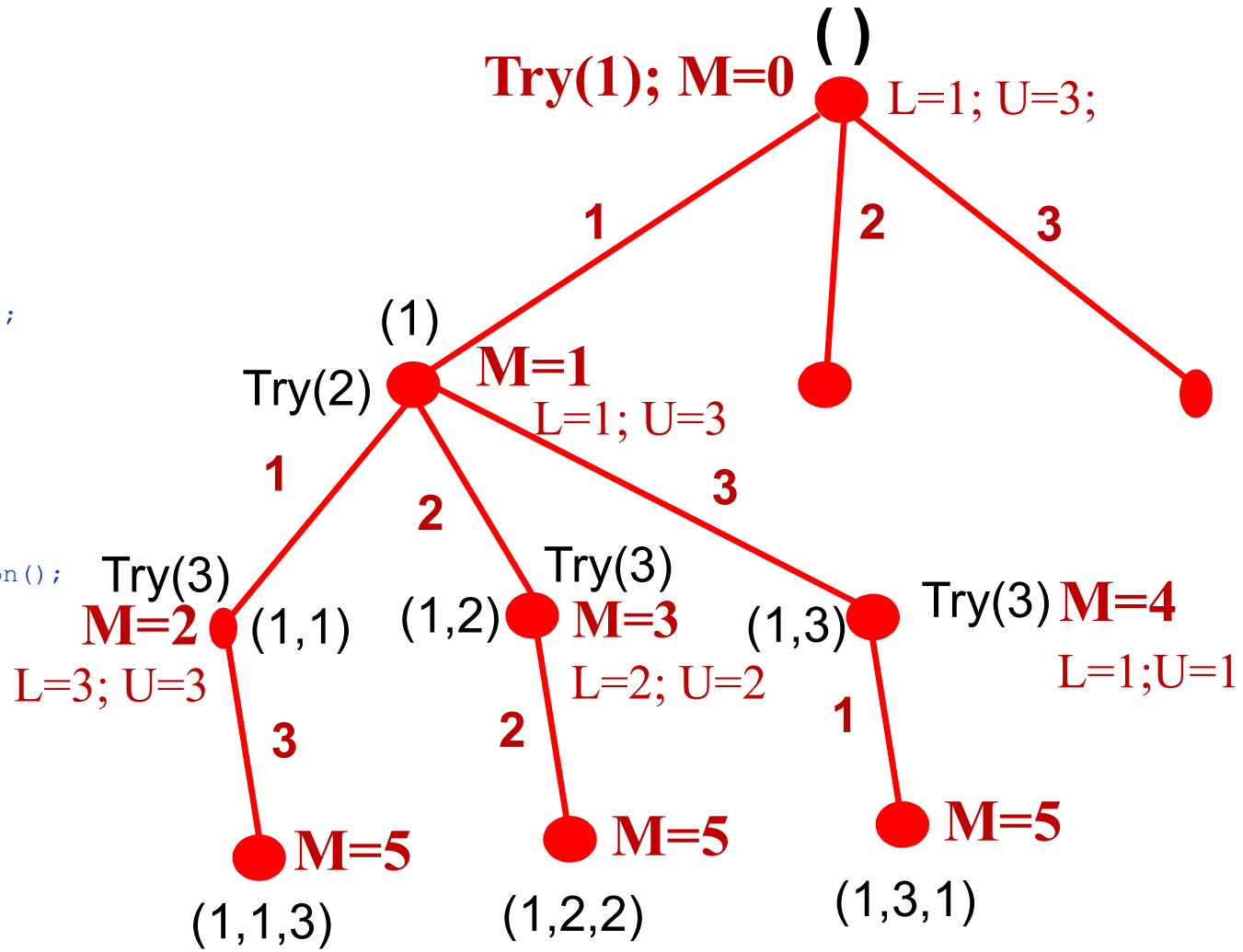
```
void Try(int i) //find x[i]: value for variable x_i
{
    if (i==k) //only the last variable x_k need to determine value
    {
        U = n - M; L = U;
    }
    else
    {
        U = n - M - (k-i); L = 1;
    }
    for (j =  L; j <= U; j++)
    {
        x[i] = j;
        M = M + j;
        if (i == k) PrintSolution();//if we know all values of k variables x_i then print solution
        else  Try(i+1);//continue to determine value for x_{i+1}
        M = M - j;
    }
}
void Problem(int k, int n)
{   M = 0; //store the sum of all variables whose values have been determined
    Try(1);
}
```

- Assume we already have partial solution $(x_1, x_2, \ldots, x_{i-1})$: it means already know values of $(i\text{-}1)^{th}$ first variables $x_1, x_2, \ldots, x_{i-1}$
- Now we build the i^{th} component of solution: it means we determine value for variable x_i. We do as following:
  - Calculate the sum of $(i\text{-}1)$ first variables that already know values: $M = x_1 + x_2 + \ldots + x_{i-1}$
  - Sum of $(k\text{-}i)$ variables $x_{i+1}, \ldots, x_k$ has the least value = $k - i$
  - Maximum value of x_i could be: $U = n - M - (k - i)$

  ➔ variable x_i can only get the value in range: $1 \le x_i \le U$

Decision tree n =5, k = 3

```
void Try(int i)
{
    if (i==k) {
        U = n - M; L = U;
    }
    else
    {
        U = n - M - (k-i); L = 1;
    }
    for (j =  L; j <= U; j++)
    {
        x[i] = j;
        M = M + j;
        if (i == k) PrintSolution();
        else  Try(i+1);
        M = M - j;
    }
}
```



Exercise

Given set X = {1, 2, 3, 4, 5, 6, 7}.

- Assume we have a 5-element subset of X: (1, 2, 5, 6, 7), find its next subset in dictionary order.
  - Scan from the right to the left of sequence $a_1, a_2, \ldots, a_m$: find the first element $a_i \neq n\text{-}m\text{+}i$
  - Replace a_i by $a_i + 1$
  - Replace a_j by $a_i + j - i$, where $j = i+1, i+2, \ldots, m$
- Assume we have permutation (2, 4, 3, 7, 6, 5, 1), find its next permutation in dictionary order.
  - Find from the left to right: 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 .