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

Data structure and Algorithm

Nguyễn Khánh Phương

Computer Science department
School of Information and Communication technology
E-mail: phuongnk@soict.hust.edu.vn

Course outline

Chapter 1. Fundamentals

Chapter 2. Algorithmic paradigms

Chapter 3. Basic data structures

Chapter 4. Tree

Chapter 5. Sorting

Chapter 6. Searching

Chapter 7. Graph

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

Chapter 2. Algorithmic paradigms

Nguyễn Khánh Phương

Computer Science department
School of Information and Communication technology
E-mail: phuongnk@soict.hust.edu.vn

Contents

Contents

1. Brute force (Exhaustive search)

Brute force – Exhaustive search:

- When the problem requires finding objects that satisfy a certain properties from a given se of projects, we can apply the brute force:
  - Browse all the objects: for each object, check whether it satisfies the required properties or not, if so, that object is the solution to find, if not, keep searching.

Example (Traveling Salesman Problem): A tourist wants to visit n cities T_1, T_2, ..., T_n. *Itinerary is a way of going from a certain city through all the remaining cities, each city exactly once, and then back to the starting city. Let* d_{ij} the cost of going fron T_i to T_j ($i, j = 1, 2,..., n$). Find itinerary with minimum total cost.

Answer: browse all $n!$ possible itineraries, each itinerary calculates the corresponding trip cost, and compares these $n!$ values to get the one with minimum value.

- Brute force: simple, but computation time is inefficient.

NGUYỄN KHÁNH PHƯƠNG
Bộ môn KHMT – ĐHBK HN

Example: Stock span problem

This problem is often asked in interviews of Google and Amazon:

> Given a list of prices of a single stock for N number of days, find stock span for each day. Stock span is defined as a number of consecutive days prior to the current day when the price of a stock was less than or equal to the price at current day.

Example:

Prices of stock in period of 6 days are {100, 60, 70, 65, 80, 85}, then stock span = {0,0,1,0,3,4}.

Solve by brute force:

Browse for each day *i* (from left to right `for i = 0,..,5`):

- `span[i] = 0;`
- scan each previous day *j* of *i* (`for j = i-1,..,0`):
  - `if price[j] <= price[i] then span[i]++;`
  - `else break;`

Complexity: $O(n^2)$ where n is number of days.

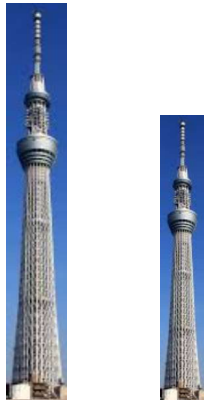
Exercise: propose algorithm with complexity of O(n) and memory O(n)

Example: Another form of stock span problem

Various signal towers are present in a city.Towers are aligned in a straight horizontal line(from left to right) and each tower transmits a signal in the right to left direction.Tower A shall block the signal of Tower B if Tower A is present to the left of Tower B and Tower A is taller than Tower B. So,the range of a signal of a given tower can be defined as :

{(the number of contiguous towers just to the left of the given tower whose height is less than or equal to the height of the given tower) + 1}.

You need to find the range of each tower.



A B

INPUT

First line contains an integer T specifying the number of test cases.

Second line contains an integer n specifying the number of towers.

Third line contains n space separated integers(H[i]) denoting the height of each tower.

OUTPUT

Print the range of each tower (separated by a space).

Constraints

1 <= T <= 10

2 <= n <= 10^6

1 <= H[i] <= 10^8

SAMPLE INPUT	SAMPLE OUTPUT
1 7 100 80 60 70 60 75 85	1 1 1 2 1 4 6

Contents

2. Recursion

2.1. The concept of recursion

2.2. Recursive algorithm diagram

2.3. Some illustrative examples

2.4. Analysis of recursive algorithm

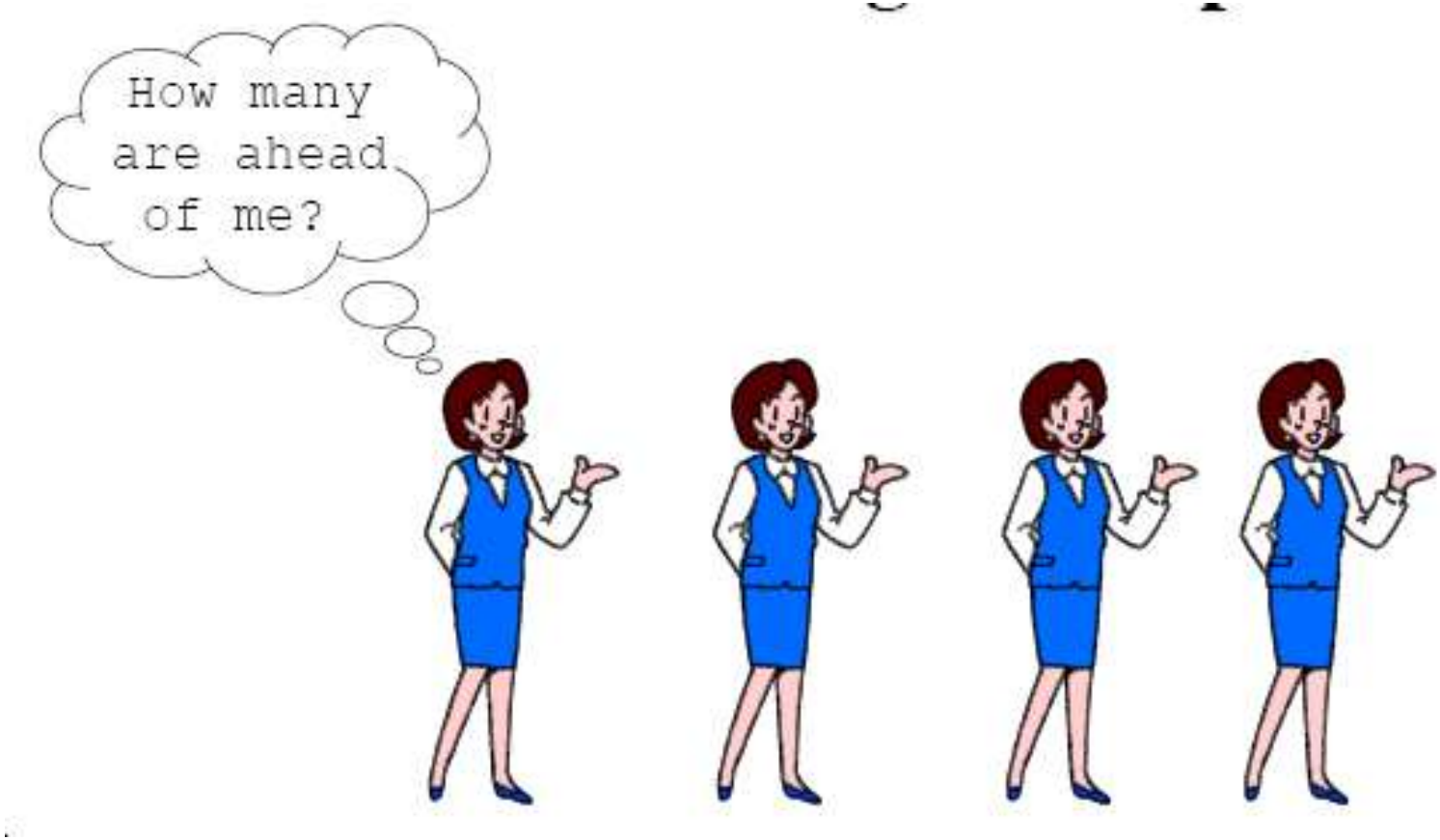
2.5. Recursion and memorization

2.1. The concept of recursion

- In practice, we often encounter objects that include themselves or are defined in terms of themselves. We say those objects are defined recursively.
- **Example**
  - Check attendance
  - Fractal
  - Recursive function
  - Sets are defined recursively
  - Trees are defined recursively
  - ...

NGUYỄN KHÁNH PHƯƠNG
Bộ môn KHMT – ĐHBK HN

Recursive example: Check attendance

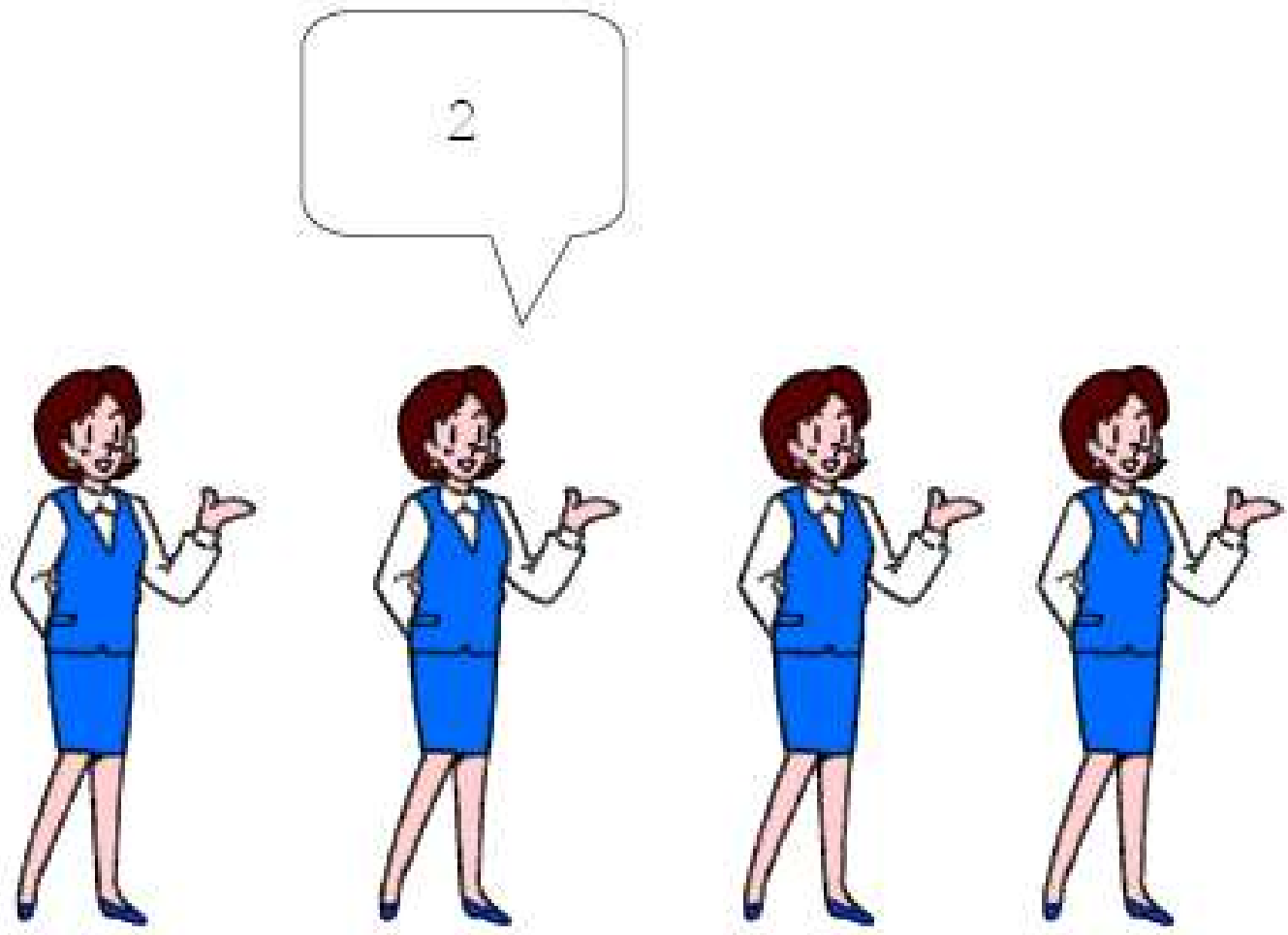


NGUYỄN KHÁNH PHƯƠNG
Bộ môn KHMT – ĐHBK HN

Recursive example: Check attendance

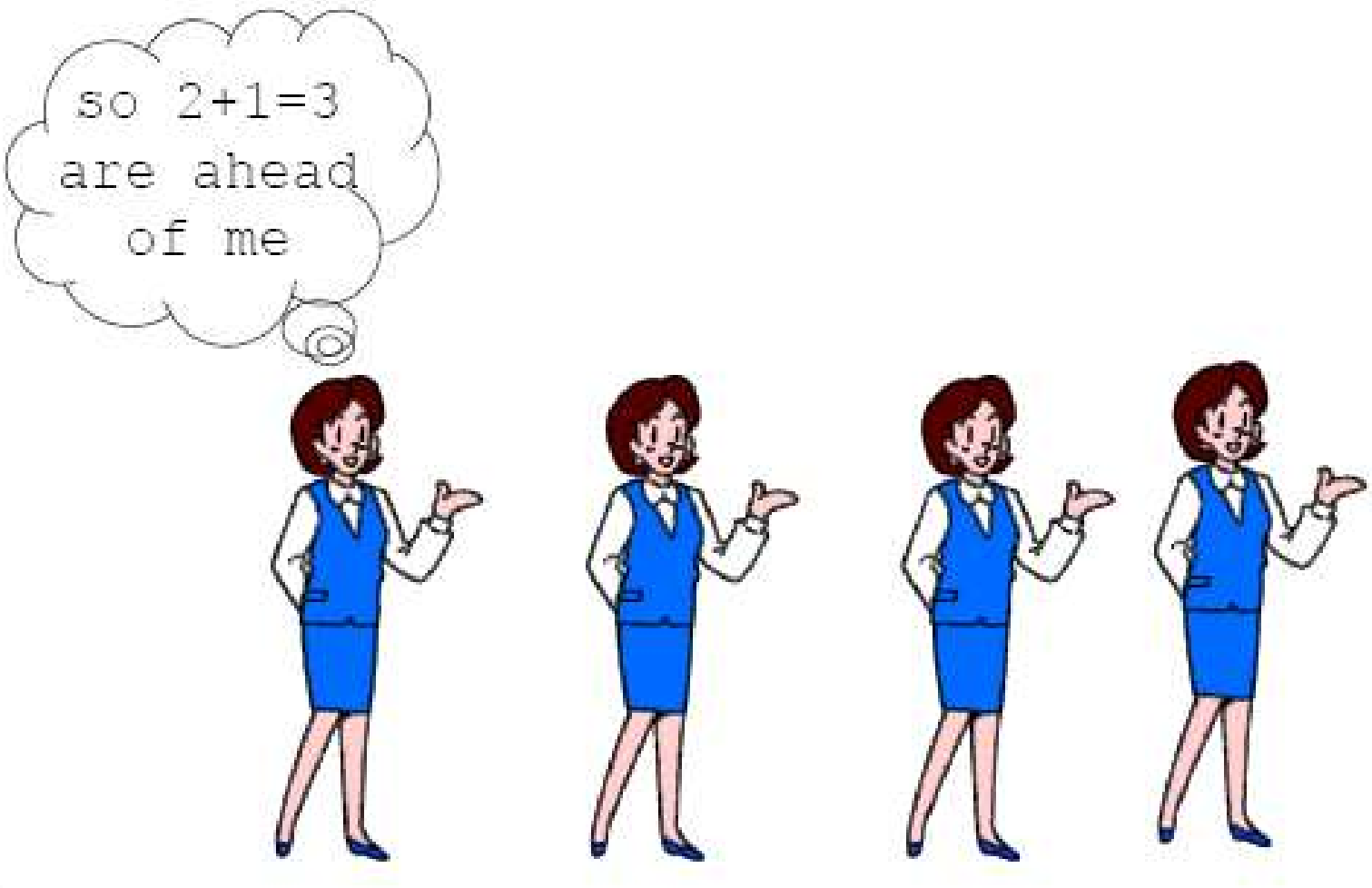
NGUYỄN KHÁNH PHƯƠNG
Bộ môn KHMT – ĐHBK HN

Recursive example: Check attendance



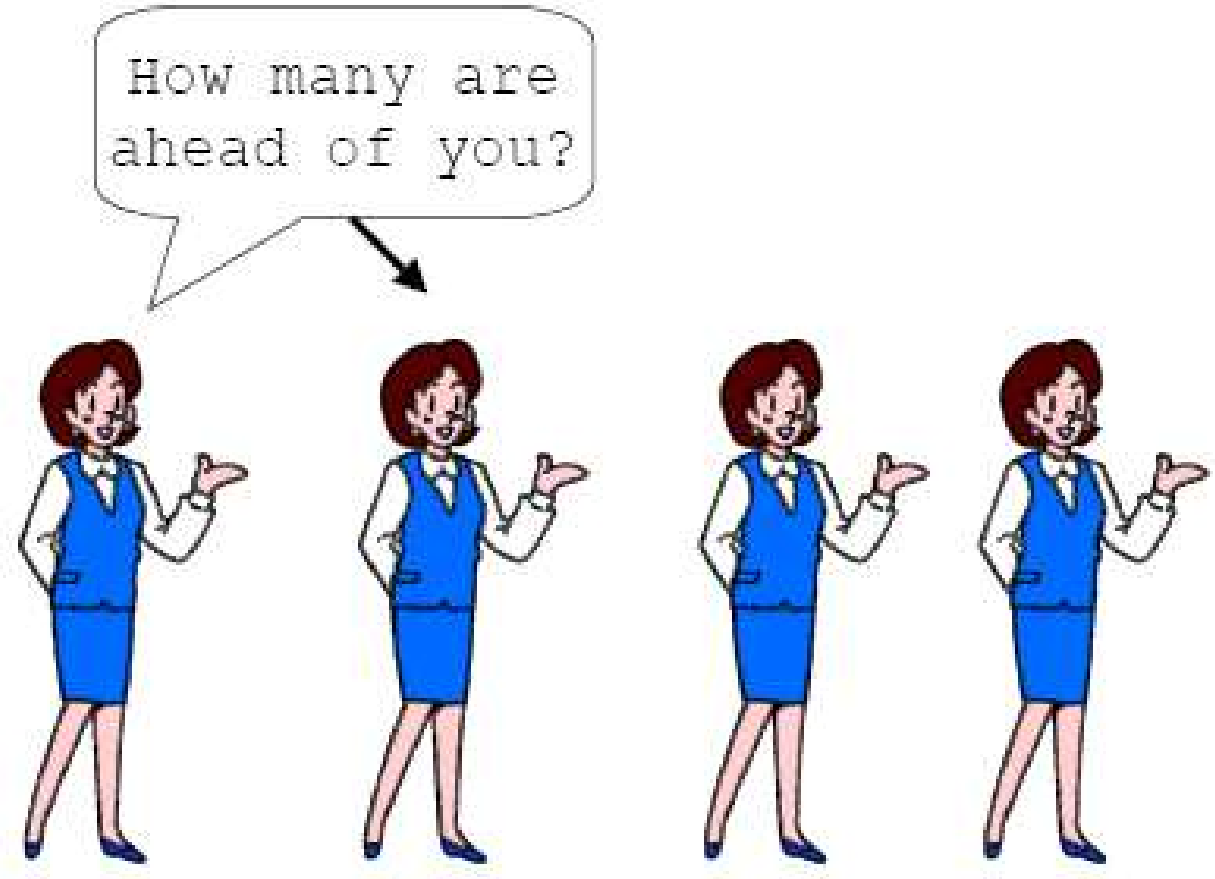
NGUYỄN KHÁNH PHƯƠNG
Bộ môn KHMT – ĐHBK HN

Recursive example: Check attendance



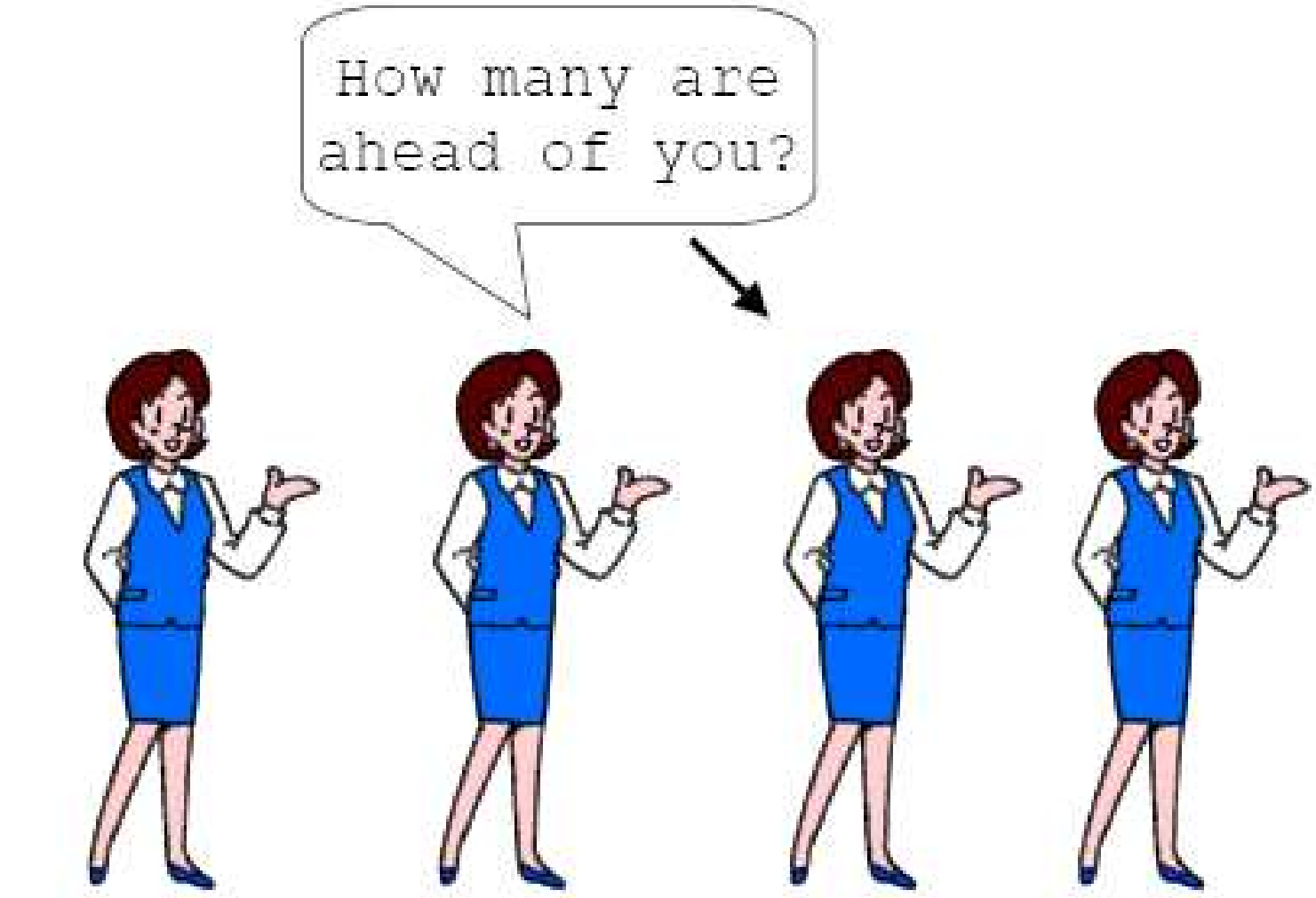
NGUYỄN KHÁNH PHƯƠNG
Bộ môn KHMT – ĐHBK HN

Recursive example: Check attendance



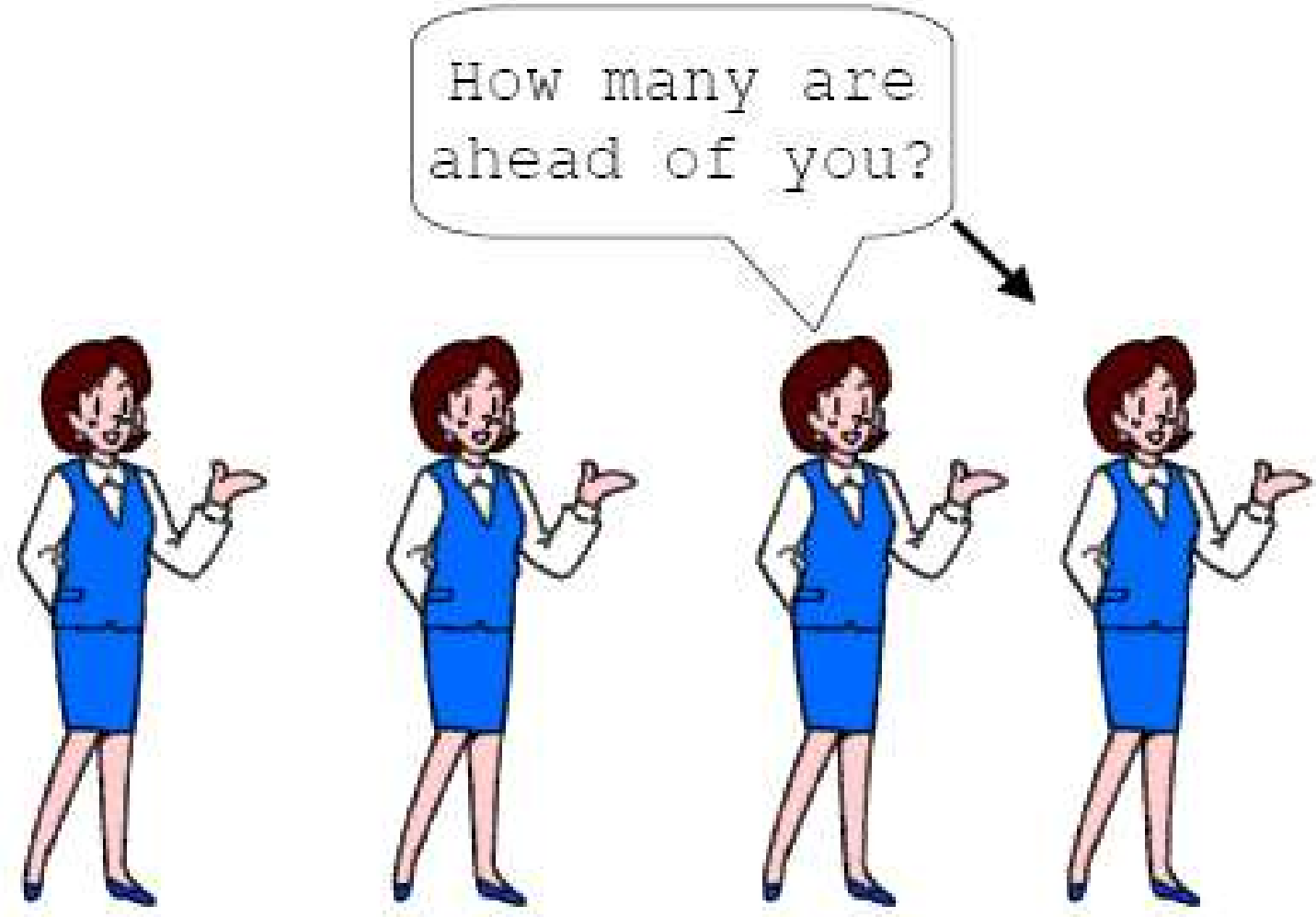
NGUYỄN KHÁNH PHƯƠNG
Bộ môn KHMT – ĐHBK HN

Recursive example: Check attendance



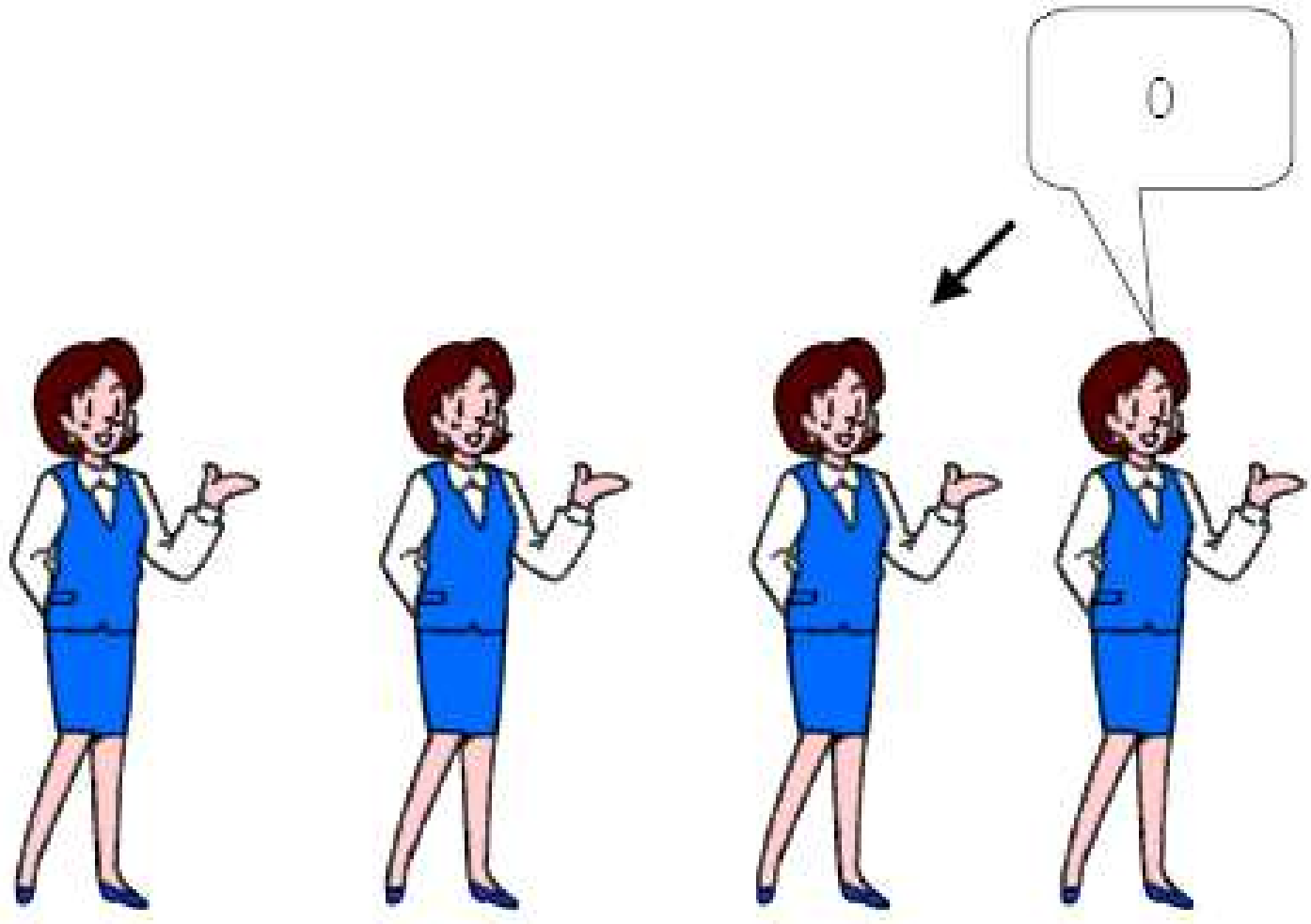
NGUYỄN KHÁNH PHƯƠNG
Bộ môn KHMT – ĐHBK HN

Recursive example: Check attendance



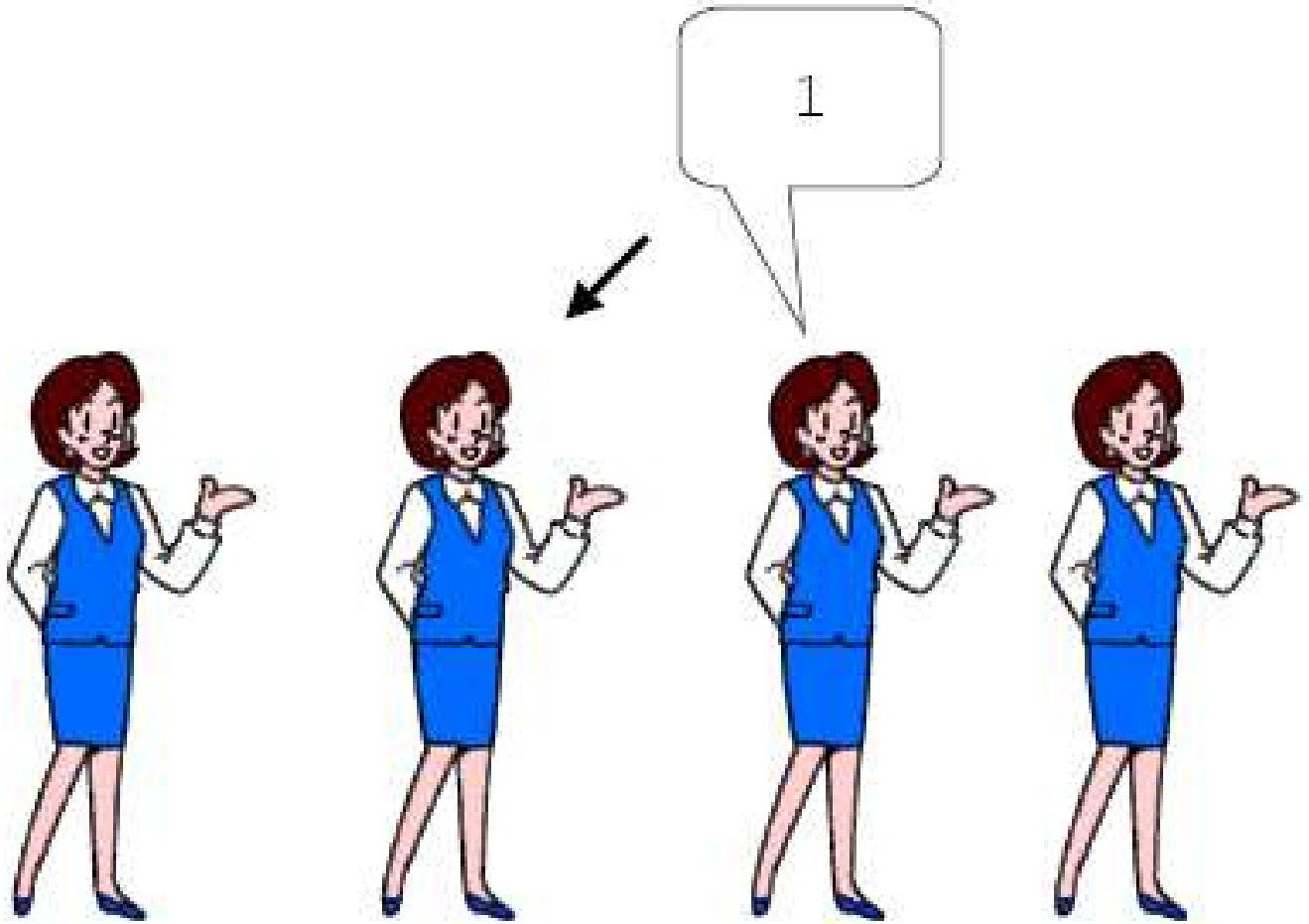
NGUYỄN KHÁNH PHƯƠNG
Bộ môn KHMT – ĐHBK HN

Recursive example: Check attendance



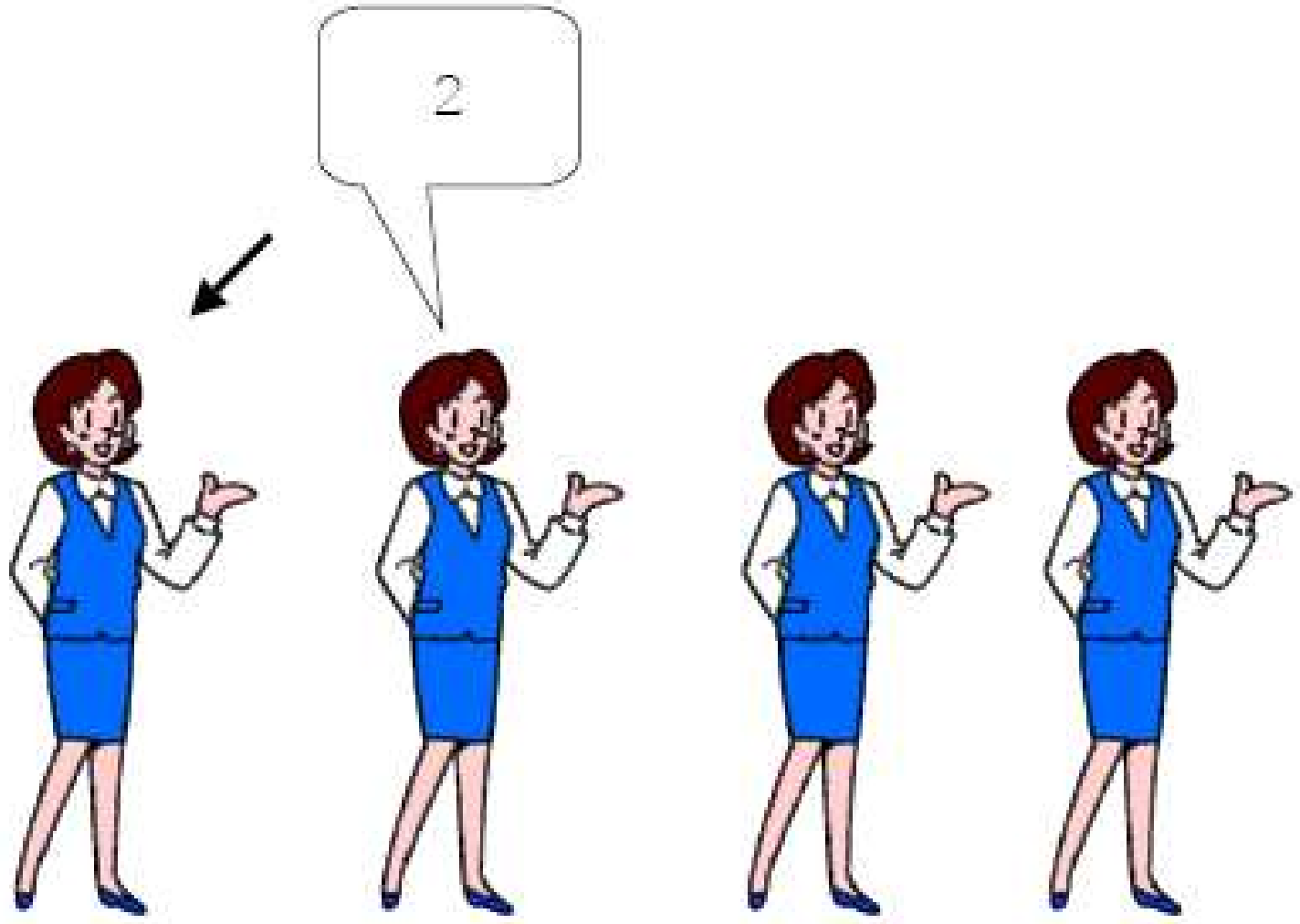
NGUYỄN KHÁNH PHƯƠNG
Bộ môn KHMT – ĐHBK HN

Recursive example: Check attendance



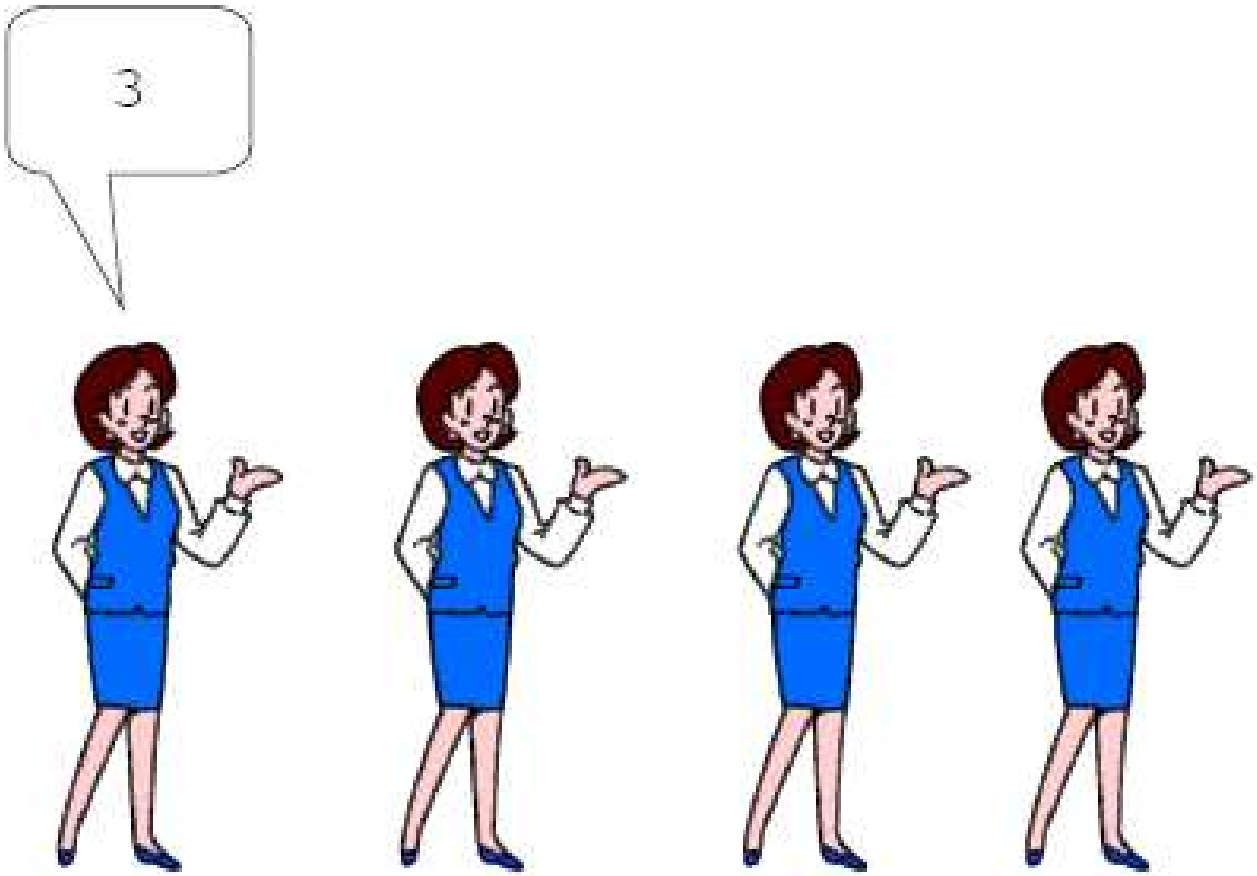
NGUYỄN KHÁNH PHƯƠNG
Bộ môn KHMT – ĐHBK HN

Recursive example: Check attendance



NGUYỄN KHÁNH PHƯƠNG
Bộ môn KHMT – ĐHBK HN

Recursive example: Check attendance

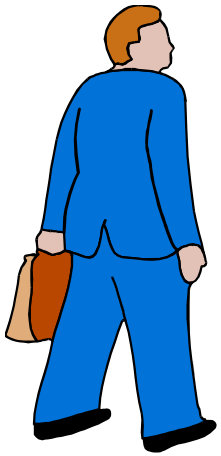


NGUYỄN KHÁNH PHƯƠNG
Bộ môn KHMT – ĐHBK HN

Example: The Handshake Problem

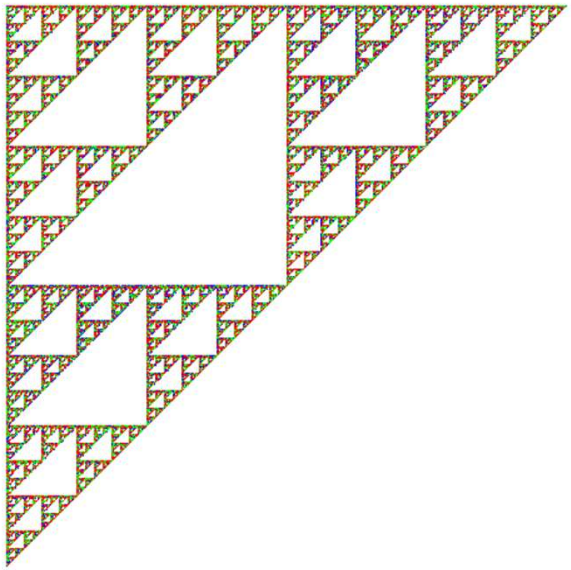
There are n people in the room. Everyone shakes hands with n-1 others, each exactly once. Calculate h(n) the total number of possible handshakes

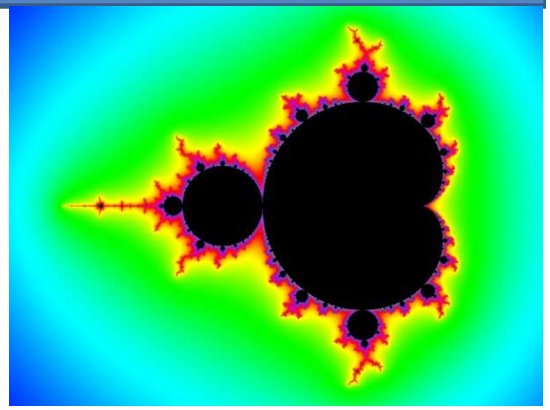
h(n) = h(n-1) + n-1 h(4) = h(3) + 3 h(3) = h(2) + 2 h(2) = 1



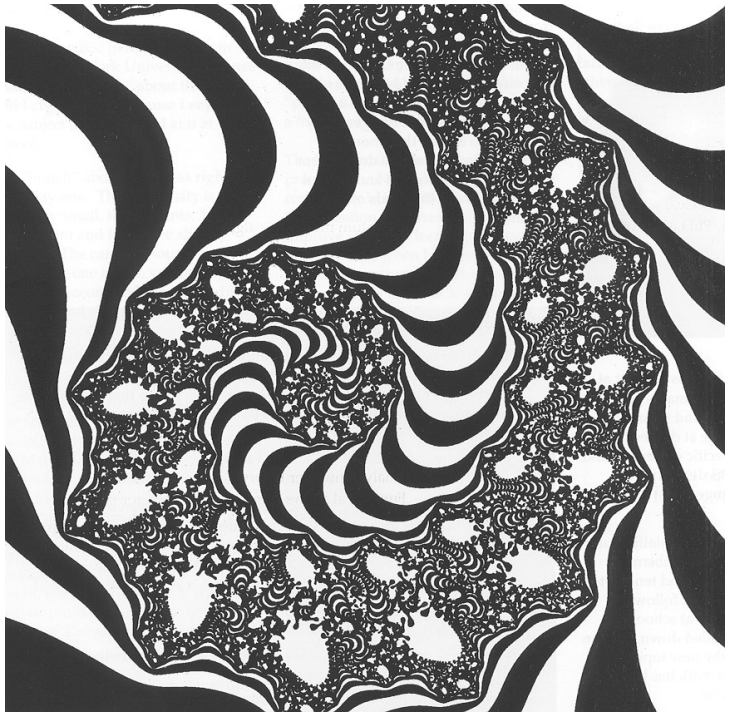
h(n): The sum of integers from 1 to n-1 = n(n-1) / 2

Recursive example: Fractals





Fractals are examples of recursively constructed images (objects that repeat recursively).



Recursive Functions

Recursive functions are determined depending on the non-negative integer variables n according to the following diagram:

Basic Step: *Determine the value of function when* n=0: $f(0)$.

Recursive Step: *Given values of* $f(k)$, $k \leq n$, *give the rules to calculate the value of* $f(n+1)$.

Example 1:

$$f(0) = 3, \qquad n = 0$$

$$f(n+1) = 2f(n) + 3, \qquad n > 0$$

Then we have: $f(1) = 2 \times 3 + 3 = 9$, $f(2) = 2 \times 9 + 3 = 21$, ...

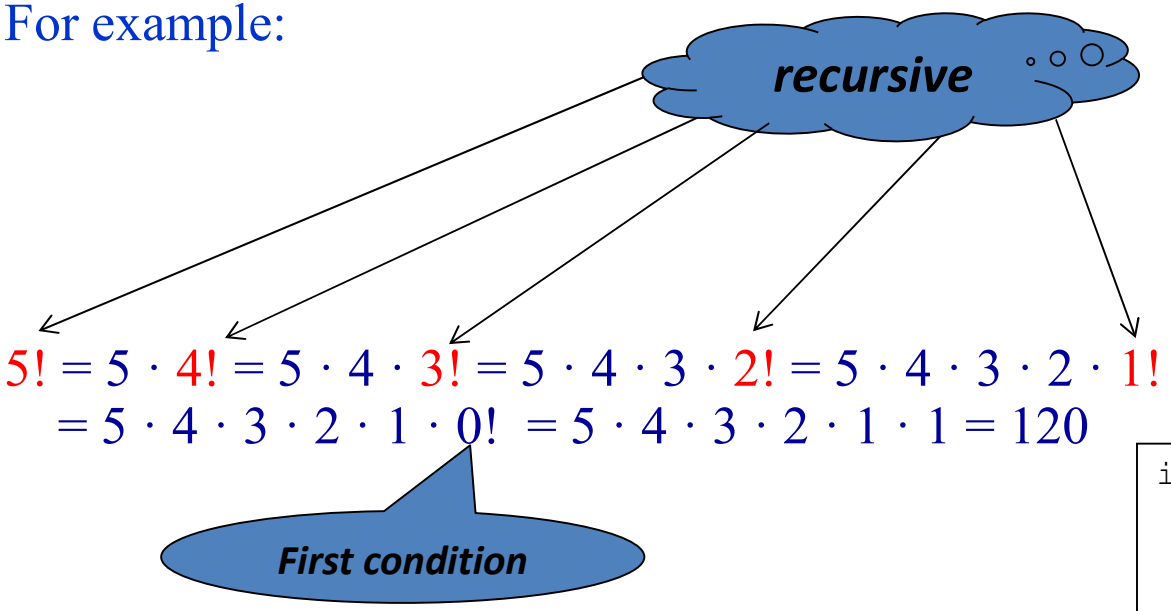
Recursive Functions

Example 2: Recursive definition to calculate $n!$

$$f(0) = 1$$
$$f(n) = n * f(n\text{-}1)$$

To calculate the value of recursive function, we replace it gradually according to the recursive definition to obtain the expression with smaller and smaller arguments until we get the first condition.

For example:




```
int factorial(int n){
 if (n==0)
     return  1;
 else
     return n*factorial(n-1);
}
```

factorial(4);

factorial(4)

```
int factorial(int n){
 if (n==0)
     return  1;
 else
     return n*factorial(n-1);
}
```

factorial(4);

factorial(4)

```
int factorial(int n){
 if (n==0)
     return  1;
 else
     return n*factorial(n-1);
}
```

n=4
Returns 4*factorial(3)

factorial(4);

factorial(4)

```
int factorial(int n){
 if (n==0)
     return  1;
 else
     return n*factorial(n-1);
}
```

n=4
Returns 4*factorial(3)

n=3
Returns 3*factorial(2)

factorial(4);

factorial(4)

```
int factorial(int n){
 if (n==0)
     return  1;
 else
     return n*factorial(n-1);
}
```

n=4
Returns 4*factorial(3)

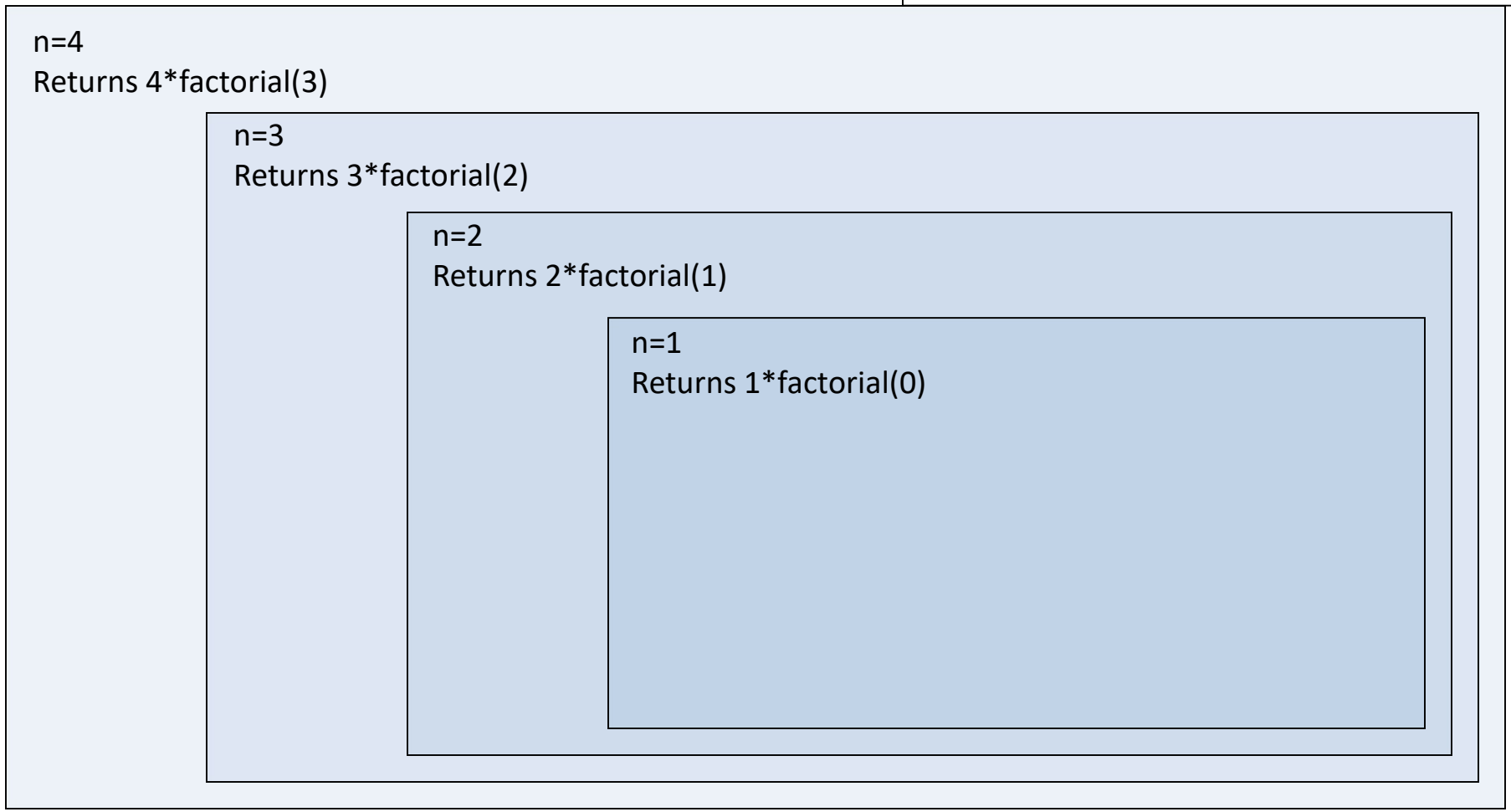
n=3
Returns 3*factorial(2)

n=2
Returns 2*factorial(1)

factorial(4);

factorial(4)

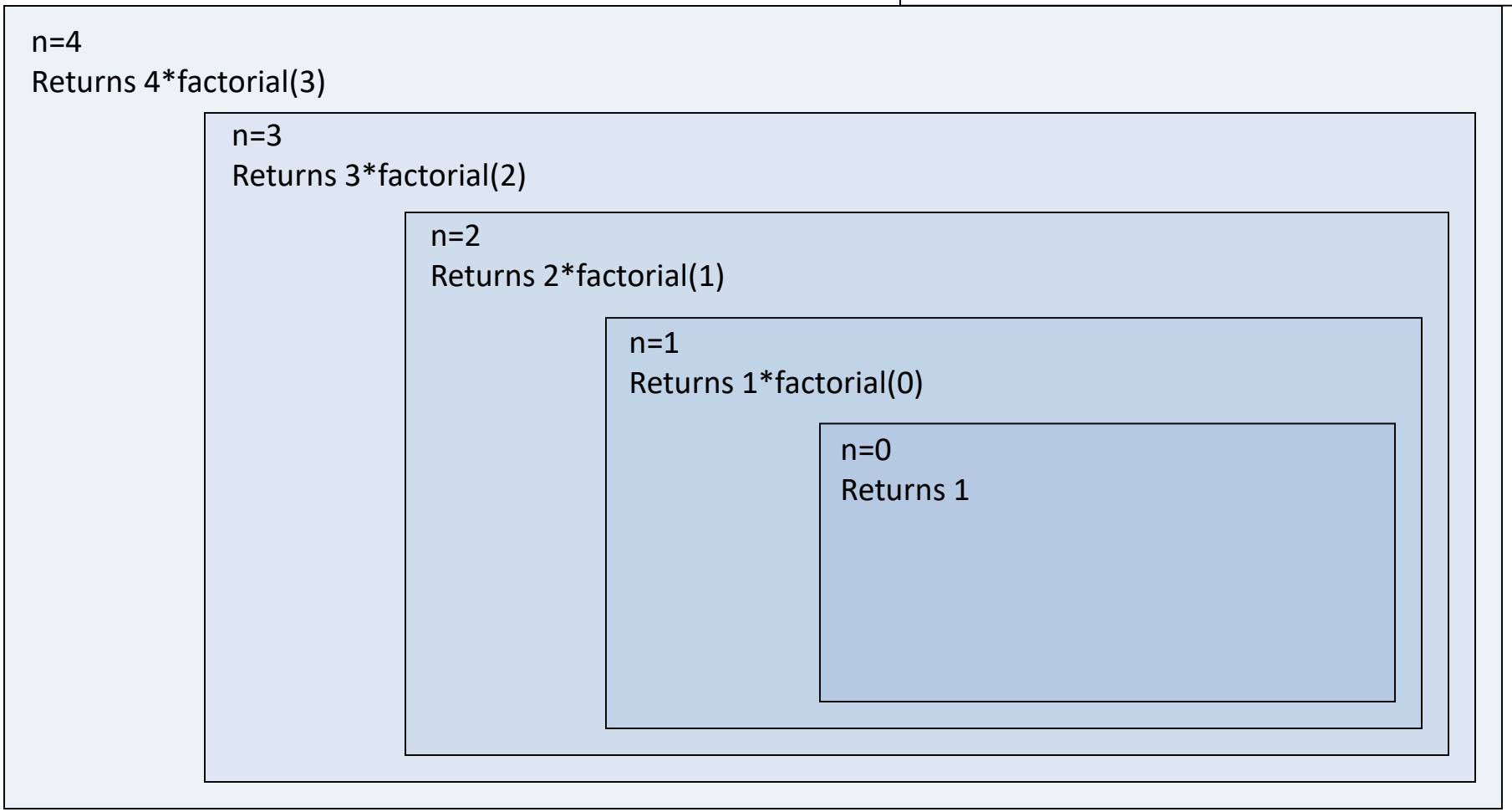
```
int factorial(int n){
 if (n==0)
     return  1;
 else
     return n*factorial(n-1);
}
```



factorial(4);

factorial(4)

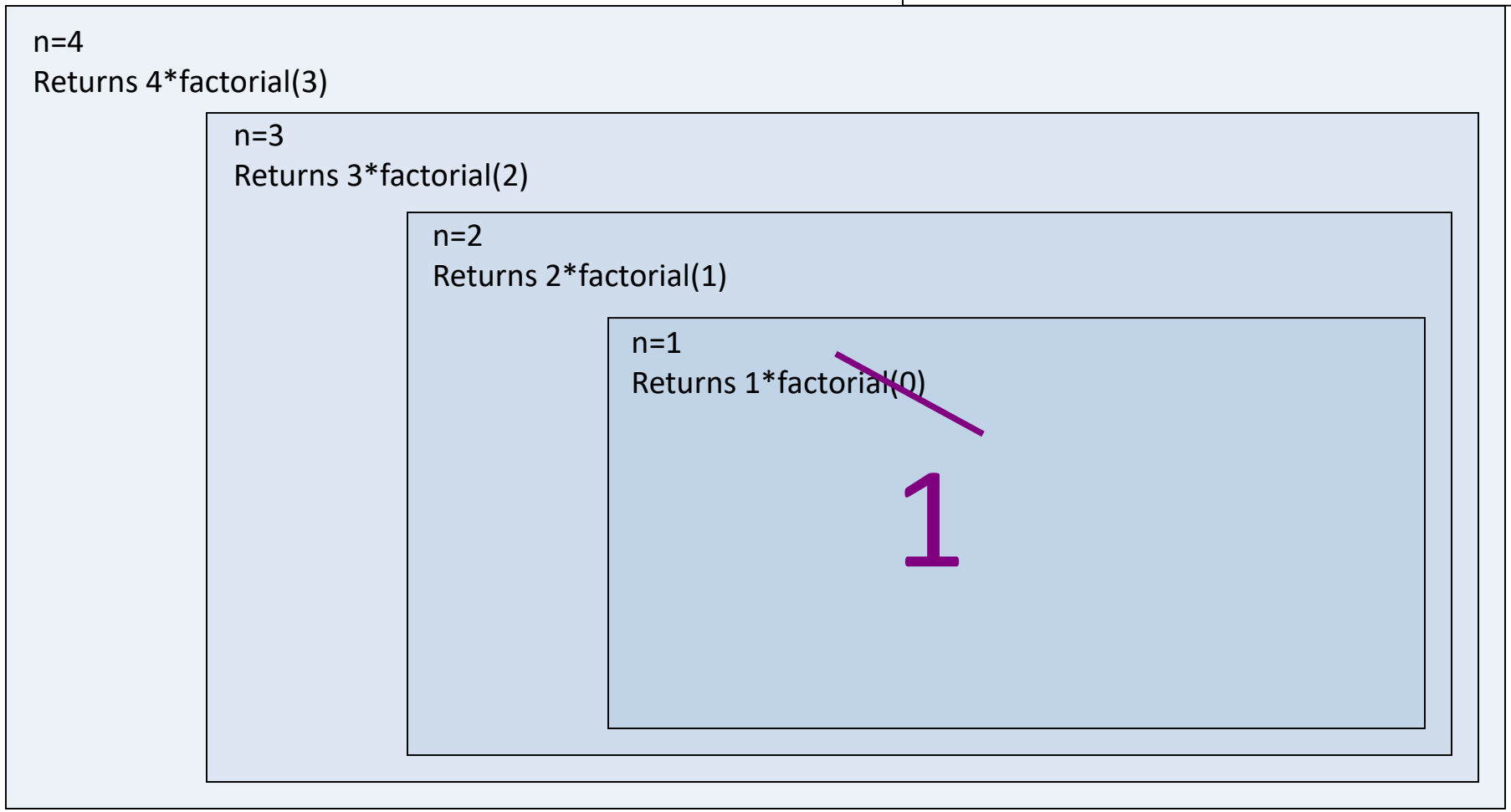
```
int factorial(int n){
 if (n==0)
     return  1;
 else
     return n*factorial(n-1);
}
```



factorial(4);

factorial(4)

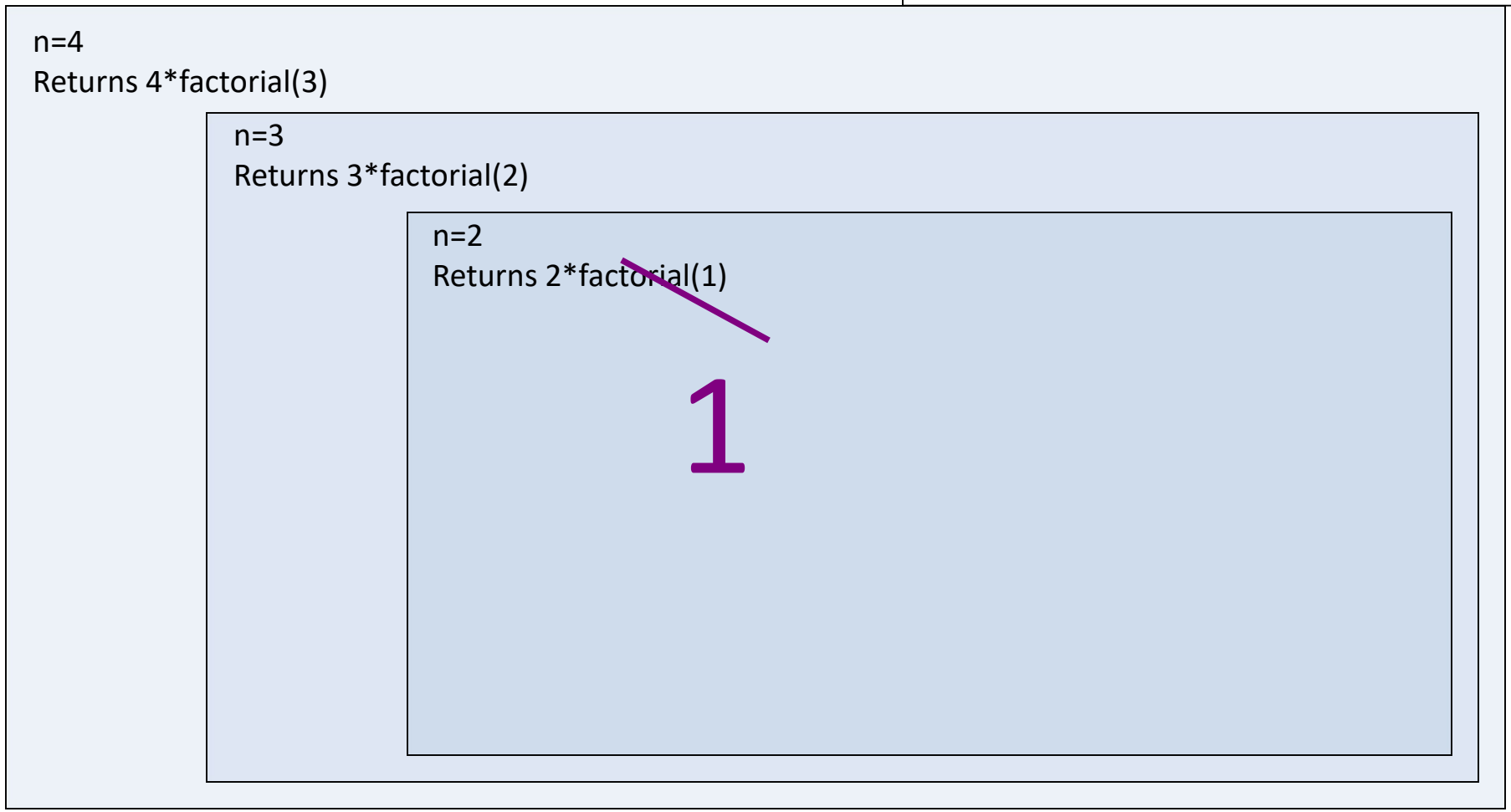
```
int factorial(int n){
 if (n==0)
     return  1;
 else
     return n*factorial(n-1);
}
```



factorial(4);

factorial(4)

```
int factorial(int n){
 if (n==0)
     return  1;
 else
     return n*factorial(n-1);
}
```



factorial(4);

factorial(4)

```
int factorial(int n){
 if (n==0)
     return  1;
 else
     return n*factorial(n-1);
}
```

n=4
Returns 4*factorial(3)

n=3
Returns 3*factorial(2)

2

factorial(4);

factorial(4)

```
int factorial(int n){
 if (n==0)
     return  1;
 else
     return n*factorial(n-1);
}
```

n=4

Returns 4*factorial(3)

6

factorial(4);

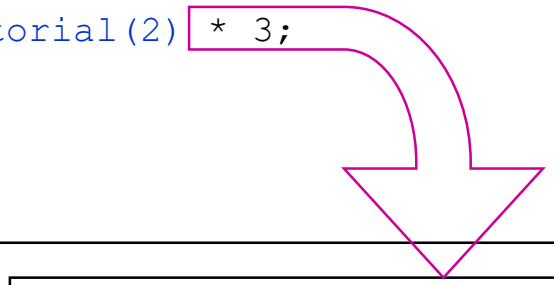
~~factorial(4)~~

24

```
int factorial(int n){
 if (n==0)
     return  1;
 else
     return n*factorial(n-1);
}
```

```
int factorial(int 3)
{
  int fact;
  if (n > 1)
    fact = factorial(2) * 3;
  else
    fact = 1;
  return fact;
}
```

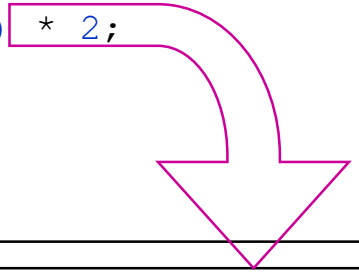
```
int factorial(int 3)
{
  int fact;
  if (n > 1)
    fact = factorial(2) * 3;
  else
    fact = 1;
  return fact;
}
```



```
int factorial(int 2)
{
  int fact;
  if (n > 1)
    fact = factorial(1) * 2;
  else
    fact = 1;
  return fact;
}
```

```
int factorial(int 3)
{
  int fact;
  if (n > 1)
    fact = factorial(2) * 3;
  else
    fact = 1;
  return fact;
}
```

```
int factorial(int 2)
{
  int fact;
  if (n > 1)
    fact = factorial(1) * 2;
  else
    fact = 1;
  return fact;
}
```

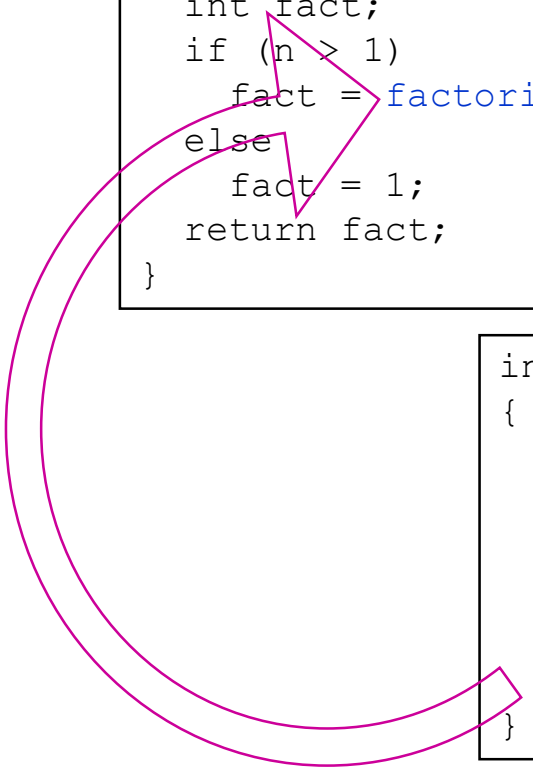


```
int factorial(int 1)
{
  int fact;
  if (n > 1)
    fact = factorial(n - 1) * n;
  else
    fact = 1;
  return fact;
}
```

```
int factorial(int 3)
{
  int fact;
  if (n > 1)
    fact = factorial(2) * 3;
  else
    fact = 1;
  return fact;
}
```

```
int factorial(int 2)
{
  int fact;
  if (n > 1)
    fact = factorial(1) * 2;
  else
    fact = 1;
  return fact;
}
```

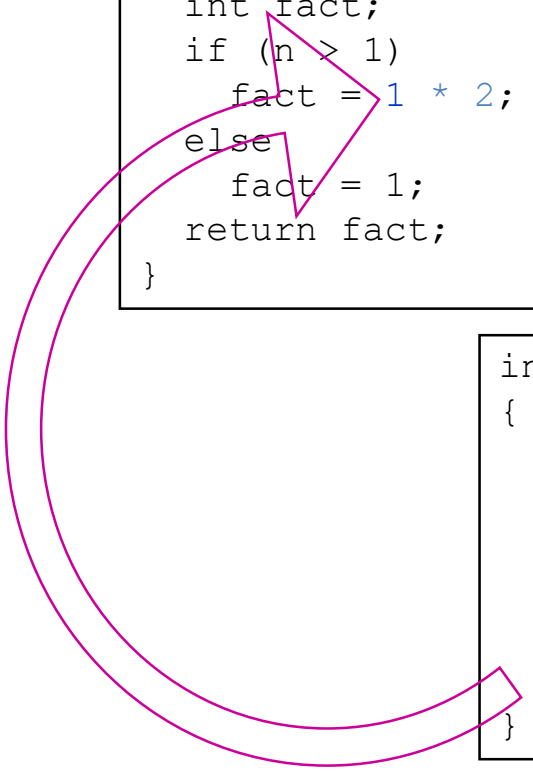
```
int factorial(int 1)
{
  int fact;
  if (n > 1)
    fact = factorial(n - 1) * n;
  else
    fact = 1;
  return 1;
}
```



```
int factorial(int 3)
{
  int fact;
  if (n > 1)
    fact = factorial(2) * 3;
  else
    fact = 1;
  return fact;
}
```

```
int factorial(int 2)
{
  int fact;
  if (n > 1)
    fact = 1 * 2;
  else
    fact = 1;
  return fact;
}
```

```
int factorial(int 1)
{
  int fact;
  if (n > 1)
    fact = factorial(n - 1) * n;
  else
    fact = 1;
  return 1;
}
```



```
int factorial(int 3)
{
  int fact;
  if (n > 1)
    fact = factorial(2) * 3;
  else
    fact = 1;
  return fact;
}
```

```
int factorial(int 2)
{
  int fact;
  if (n > 1)
    fact = 1 * 2;
  else
    fact = 1;
  return 2;
}
```

```
int factorial(int 3)
{
  int fact;
  if (n > 1)
    fact = 2 * 3;
  else
    fact = 1;
  return fact;
}
```

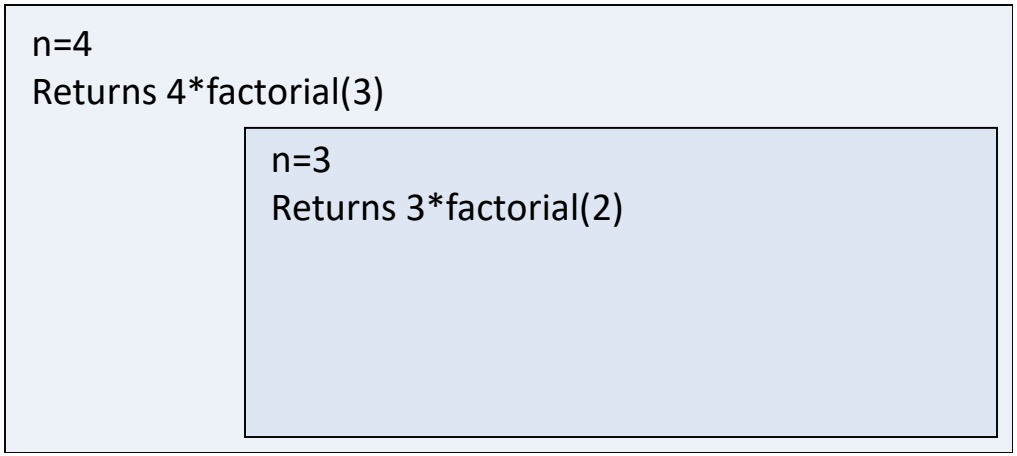
```
int factorial(int 2)
{
  int fact;
  if (n > 1)
    fact = 1 * 2;
  else
    fact = 1;
  return 2;
}
```

```
int factorial(int 3)
{
  int fact;
  if (n > 1)
    fact = 2 * 3;
  else
    fact = 1;
  return 6;
}
```

How does this program run ?

Execution of the factorial (4) function will stop until factorial (3) returns the result

When factorial (3) returns the result, the factorial (4) function continues to be executed



Recursive functions are all re-implementable using the while / for loop

- We can also implement to calculate value of *n*! by using the `while` loop

```
int factorial (int n)
{
   i=n; fact = 1;
   while(i >= 1)
   {
      fact=fact*i;
      i--;
   }
   return fact;
}
```

```
int factorial(int n){
 if (n==0)
     return  1;
 else
     return n*factorial(n-1);
}
```

Note: Replacing a recursive function by a non-recursive function is often called **recursive reduction (khử đệ quy)**. Recursive reduction is not always as easy to perform as it is in the case of factorial calculations.

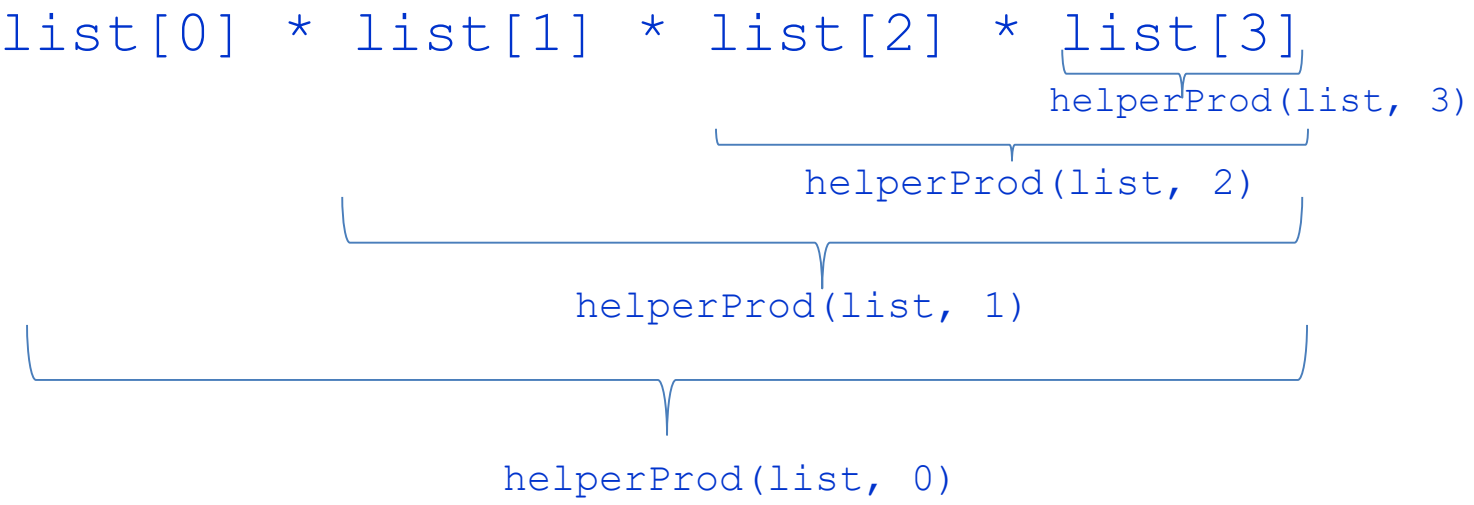
Example 3

Write a recursive function `prod(list)` to calculate the product of all numbers in the list

Example: `prod({1,3,3,4}) = 36`

Suggest: Build recursive function:

`helperProd(list, k) = list[k]*list[k+1]*…list[list.size-1]`



Example 3

Write recursive function `prod(list)` to calculate the product of all numbers in the list

Example

`prod({1,3,3,4}) = 36`

```
int listSize;
float helperProd(float list[], int k){
    if (k == listSize-1)
        return list[k];
    else
        return list[k]*helperProd(list,k+1);
}

int main()
{
    float list[] = {5, 2, 4 , 2};
    listSize = 4;
    float KQ = helperProd(list, 0);
    cout<<"Ket qua = "<<KQ<<endl;
    return 0;
}
```

Recursive Functions

***Exercise*: Write recursive function to calculate the sum of numbers**

$a_0, a_1, \ldots, a_{n-1}$

$$s_n = \sum_{k=0}^{n-1} a_k$$

Recursive definition of sum:

$$s_1 = a_0$$

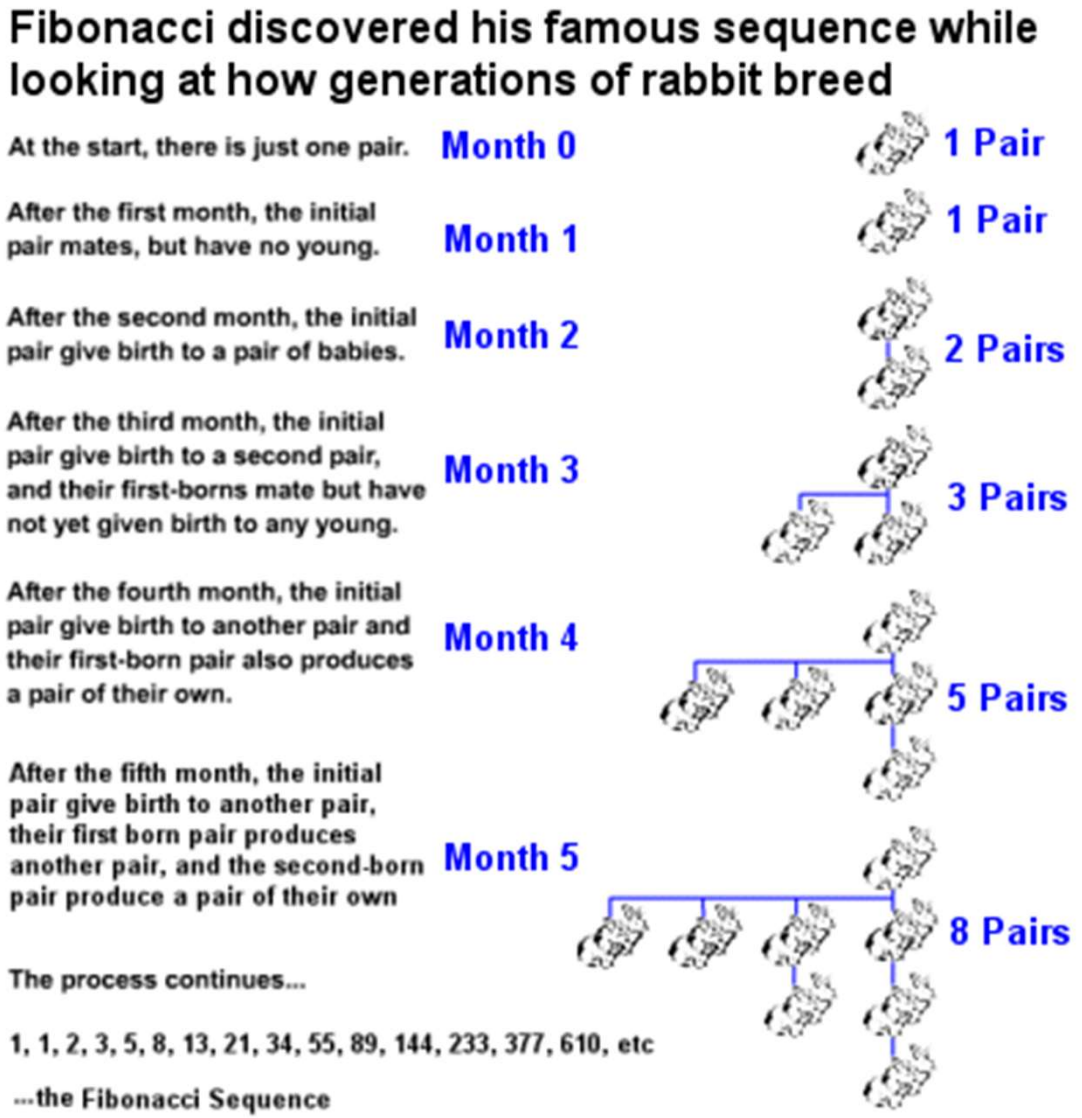
$$s_n = s_{n-1} + a_{n-1}$$

NGUYỄN KHÁNH PHƯƠNG
Bộ môn KHMT – ĐHBK HN

Example 4. Fibonacci

The farmer raised a pair of newborn rabbits. At 2 months of age, another pair of rabbits is born every month. How many pairs of rabbits are there after n months ?

- $n = 1: f_1 = 1$
- $n = 2: f_2 = 1$
- $n > 2: f_n = f_{n-1} + f_{n-2}$



Example 4. Fibonacci

Recursive function $F(n)$:

- First values : $F(n=0) = 0$; $F(n=1) = 1$
- Recursive formular: $F(n) = F(n\text{-}1) + F(n\text{-}2)$

Recursive implementation

```
int F(int n)
    if n < 2 then
        return n;
    else
        return F(n-1) + F(n-2);
```

Implement by using loop

```
int F(int n)
if n < 2 then return n;
else
{
    x= 0;          <-- F(n-2)
    y= 1;          <-- F(n-1)
    for k = 1 to n-1
    {   z = x+y;   <-- F(n)
        x = y;
        y = z;
    }
}
return y;  //y is F(n)
```

Example 5. Recursively defined sets (Tập hợp được xác định đệ qui)

The set can be recursively defined according to a diagram similar to a recursive function:

Basis Step: *define the base set (e.g. empty set).*

Recursive Step : *Define the rule to generate new set from existing sets.*

Example:

Basis Step: 3 is the element of set *S*.

Recursive Step: if *x* and y are elements of *S* then *x+y* is also an element of *S*.

3 ➔ S= {3}

3+3=6 ➔ S = {3, 6}

3+6 = 9 & 6+6=12 ➔ S = {3, 6, 9, 12}

...

Example 6. Rooted tree

Tree is an important data structure that is often used to search or sort the data

Rooted Trees(Cây có gốc): *Tree consists of nodes, with a special node called root (gốc) and edges connecting nodes.*

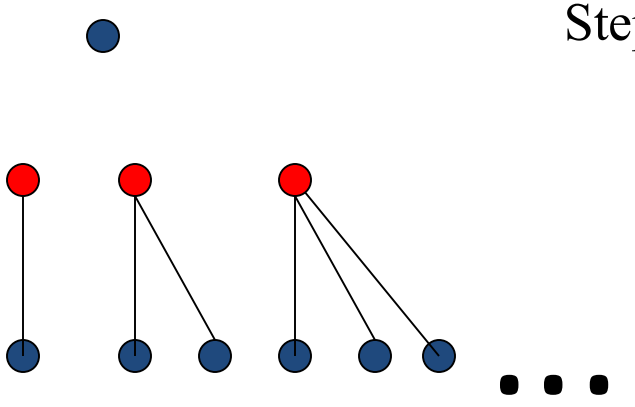
- *Basic Step: Rooted tree has only one node.*
- *Recursive Step: Assume $T_1,\ldots,T_n,\ldots$ are rooted trees with root respectively $r_1,\ldots.r_n,\ldots$*

If we create a new root r and concatenate this root with each root $r_1,\ldots r_n,\ldots$

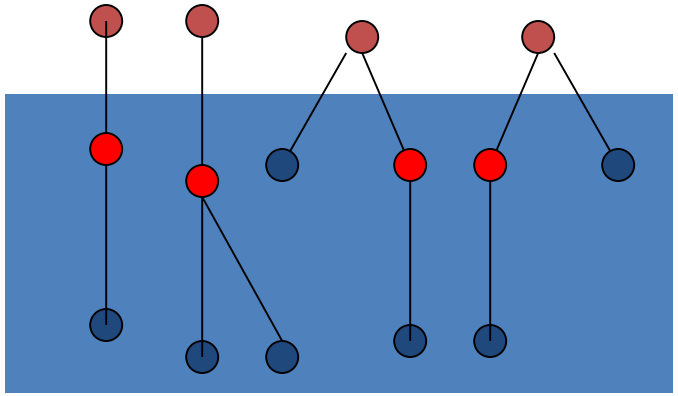
by using an edge, we obtain a new rooted tree.

Basis step:

Step 1:



Step 2:



2. Recursion

2.1. The concept of recursion

2.2. Recursive algorithm diagram

2.3. Some illustrative examples

2.4. Analysis of recursive algorithm

2.5. Recursion and memorization

Recursive algorithm diagram

```
procedure RecAlg(input)
{
    if (the size of input is minimum) then
      Do basic step;  //solve problem with smallest input size
    else
    {
      RecAlg(input with smaller size) ; // recursive step
      //could have more  than one call to RecAlg
      Combine solution of subproblem to obtain solution;
      return (solution) ;
    }
}
```

Computation time:

T(n) = if (base case) then const cost
else (time to solve all subproblems +
time to combine solutions)

Computation time depends on:

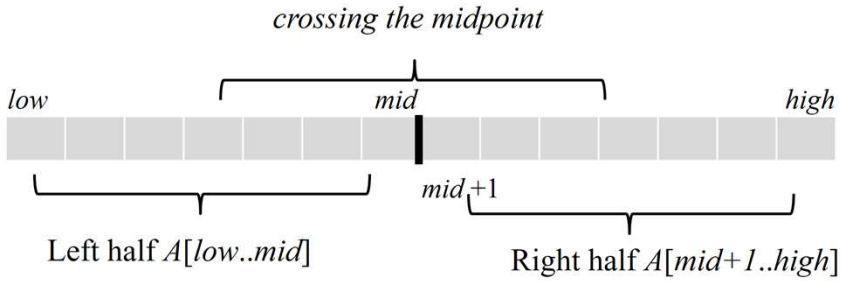
- Number of subproblems
- Size of subproblem
- Time to combine solutions of subproblems

NGUYỄN KHÁNH PHƯƠNG
Bộ môn KHMT – ĐHBK HN

Example: Recursive algorithm to solve max subarray problem

```
int MaxLeft(a, low, high)
{
  maxSum = -∞; sum = 0;
  for (int k=high; k>=low; k--) {
       sum = sum + a[k];
       maxSum = max (sum, maxSum);
  }
  return maxSum;
}
```

⬅ O(n)




```
int MaxRight(a, low, high)
{
  maxSum = -∞; sum = 0;
  for (int k=low; k<= high; k++) {
       sum = sum + a[k];
       maxSum = max (sum, maxSum);
  }
  return maxSum;
}
```

⬅ O(n)

```
int MaxSub(a, low, high)
{
  if (low == high) return a[low]; //base case: only 1 element
  else {
    int mid = (low + high) / 2;
    wL = MaxSub(a, low, mid);
    wR = MaxSub(a, mid+1, high);
    wM = MaxLeft(a, low, mid) + MaxRight(a, mid+1, high);
    return max(wL, wR, wM);
  }
}
```

⬅ T(n/2)

⬅ T(n/2)

$T(n) = 2T(n/2)+O(n)$

➔ $T(n) = O(n \log n)$

2. Recursion

2.1. The concept of recursion

2.2. Recursive algorithm diagram

2.3. Some illustrative examples

2.4. Analysis of recursive algorithm

2.5. Recursion with memorization

Example 1: Calculate the binomial coefficient

- The binomial coefficient $C(n,k)$ is defined recursively as following:

$$C(n,0) = 1,\ \ C(n,n) = 1;\quad \text{where } n >= 0,$$

$$C(n,k) = C(n\text{-}1,k\text{-}1) + C(n\text{-}1,k),\ \ \text{where } 0 < k < n$$

- Recursive implementation on C:

```
int C(int n, int k){
    if ((k==0)||(k==n)) return 1;
    else return C(n-1,k-1)+C(n-1,k);
}
```

NGUYỄN KHÁNH PHƯƠNG
Bộ môn KHMT – ĐHBK HN

Example 2: Recursive Binary Search

Input: An array *S* consists of *n* elements: S[0],…,S[*n*-1] in ascending order; Value ***key*** with the same data type as array *S*.

Output: the index in array if ***key*** is found, -1 if ***key*** is not found

Binary search algorithm: The value *key* either

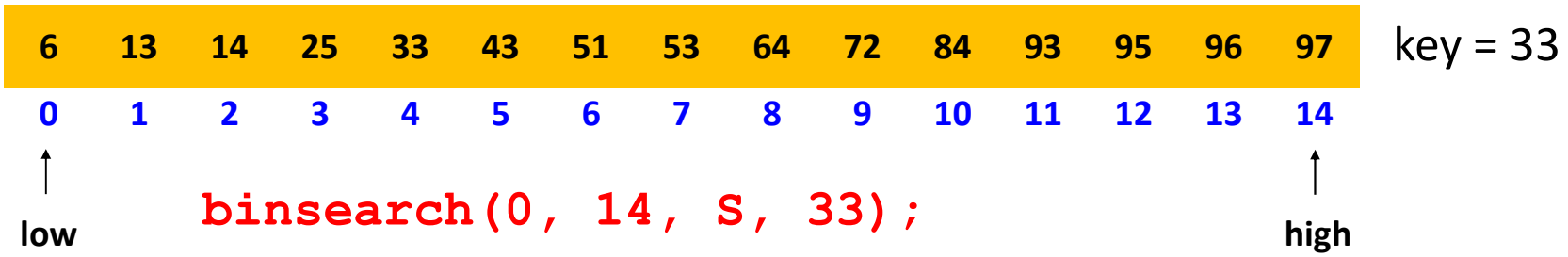
equals to the element at the middle of the array S,

or is at the left half (L) of the array S,

or is at the right half (R) of the array S.

(The situation L (R) happen only when *key* is smaller (larger) than the element at the middle of the array S)

```
int binsearch(int low, int high, int S[], int key)
```



Example 2: Recursive Binary Search

Input: An array *S* consists of *n* elements: S[0],…,S[*n*-1] in ascending order; Value ***key*** with the same data type as array *S*.

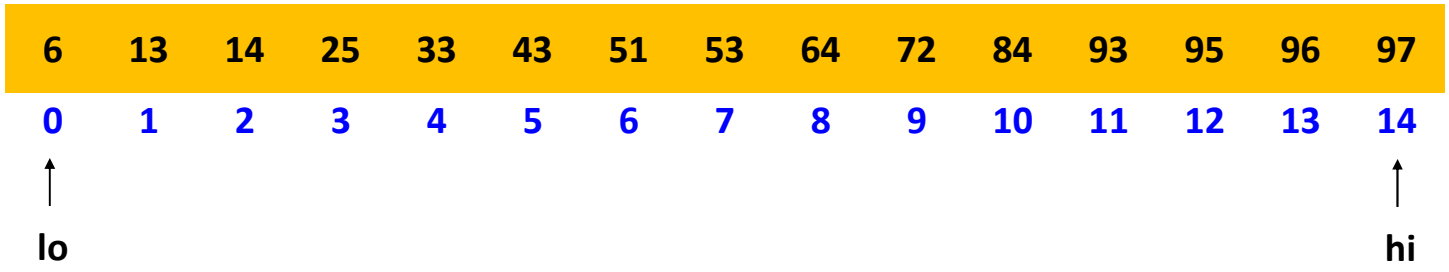
Output: the index in array if ***key*** is found, -1 if ***key*** is not found

```
int binsearch(int low, int high, int S[], int key)
{
    if (low <= high)
    {
        int mid = (low + high) / 2;
        if (S[mid]==key) return mid;
        else if (key < S[mid])
            return binsearch(low, mid-1, S, key);
        else
            return binsearch(mid+1, high, S, key);
     }
     else return -1;
}
```

Example: Binary Search

```
int binsearch(int low, int high, int S[], int key)
{
   if (low <= high)
   {
       int mid = (low + high) / 2;
       if (S[mid]== key) return mid;
       else if (key < S[mid])
           return binsearch(low, mid-1, S, key);
       else
           return binsearch(mid+1, high, S, key);
    }
    else return -1;
}
```

key=33

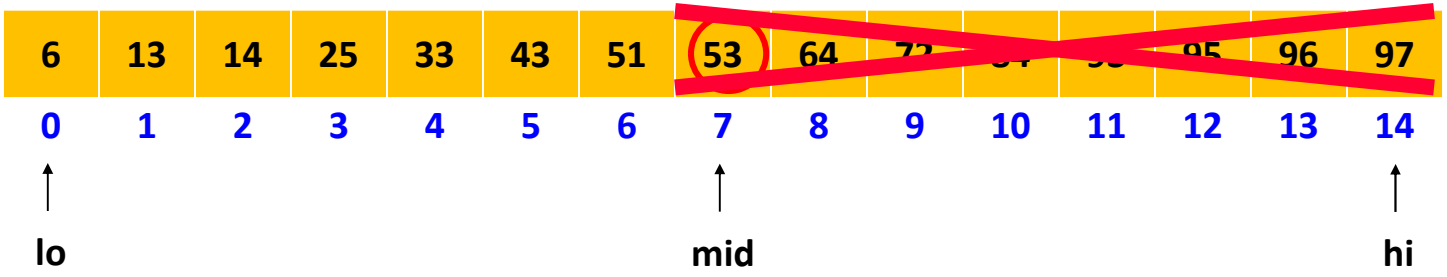


binsearch(0, 14, S, 33);

Example: Binary Search

```
int binsearch(int low, int high, int S[], int key)
{
    if (low <= high)
    {
        int mid = (low + high) / 2;
        if (S[mid]== key) return mid;
        else if (key < S[mid])
            return binsearch(low, mid-1, S, key);
        else
            return binsearch(mid+1, high, S, key);
     }
     else return -1;
}
```

key=33



binsearch(0, 14, S, 33);

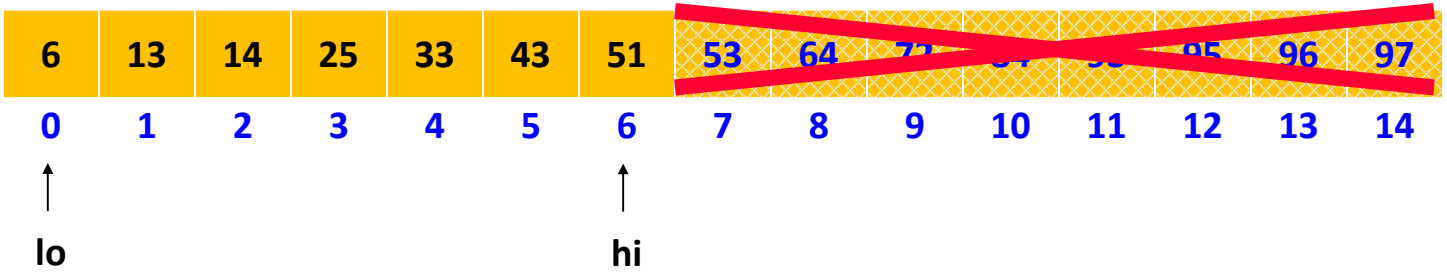
binsearch(0, 6, S, 33);

The section to be investigated is halved after each iteration

Example: Binary Search

```
int binsearch(int low, int high, int S[], int key)
{
   if (low <= high)
   {
       int mid = (low + high) / 2;
       if (S[mid]== key) return mid;
       else if (key < S[mid])
           return binsearch(low, mid-1, S, key);
       else
           return binsearch(mid+1, high, S, key);
    }
    else return -1;
}
```

key=33



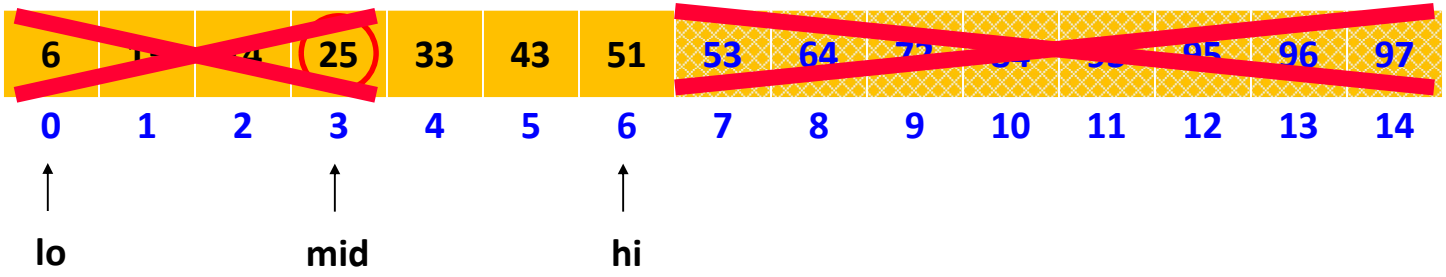
binsearch(0, 14, S, 33);

binsearch(0, 6, S, 33);

Example: Binary Search

```
int binsearch(int low, int high, int S[], int key)
{
    if (low <= high)
    {
        int mid = (low + high) / 2;
        if (S[mid]== key) return mid;
        else if (key < S[mid])
            return binsearch(low, mid-1, S, key);
        else
            return binsearch(mid+1, high, S, key);
     }
     else return -1;
}
```

key=33



binsearch(0, 14, S, 33);

binsearch(0, 6, S, 33);

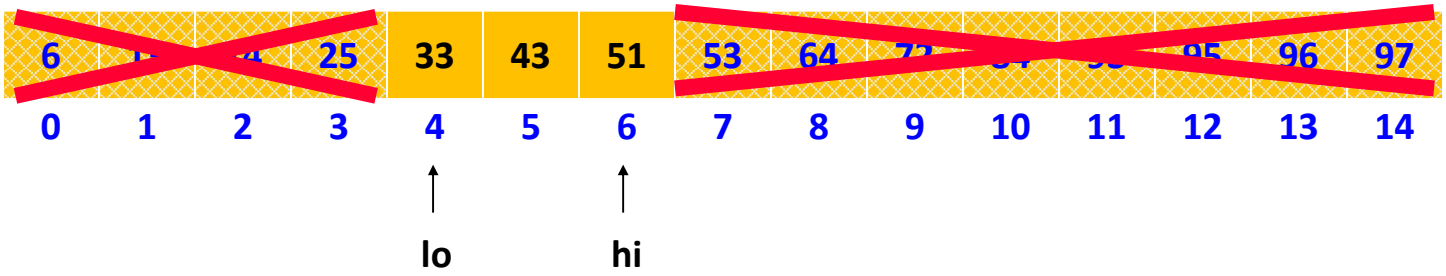
binsearch(4, 6, S, 33);

The section to be investigated is halved after each iteration

Example: Binary Search

```
int binsearch(int low, int high, int S[], int key)
{
   if (low <= high)
   {
       int mid = (low + high) / 2;
       if (S[mid]== key) return mid;
       else if (key < S[mid])
           return binsearch(low, mid-1, S, key);
       else
           return binsearch(mid+1, high, S, key);
    }
    else return -1;
}
```

key=33



binsearch(0, 14, S, 33);

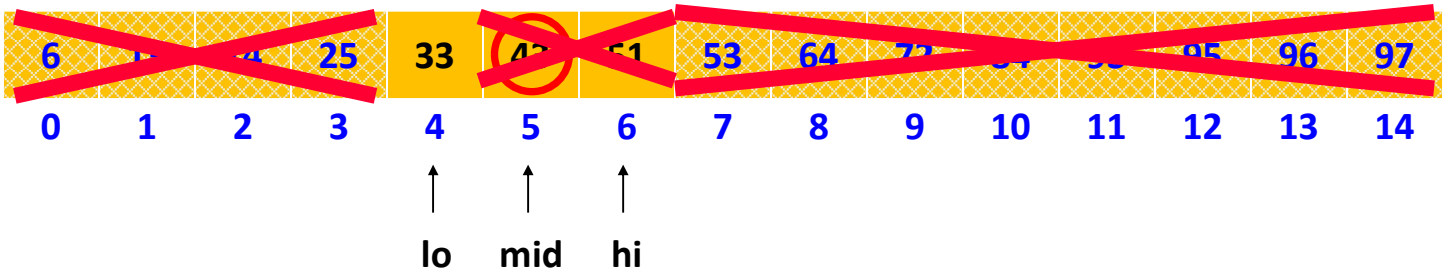
binsearch(0, 6, S, 33);

binsearch(4, 6, S, 33);

Example: Binary Search

```
int binsearch(int low, int high, int S[], int key)
{
    if (low <= high)
    {
        int mid = (low + high) / 2;
        if (S[mid]== key) return mid;
        else if (key < S[mid])
            return binsearch(low, mid-1, S, key);
        else
            return binsearch(mid+1, high, S, key);
     }
     else return -1;
}
```

key=33



binsearch(0, 14, S, 33);

binsearch(0, 6, S, 33);

binsearch(4, 6, S, 33);

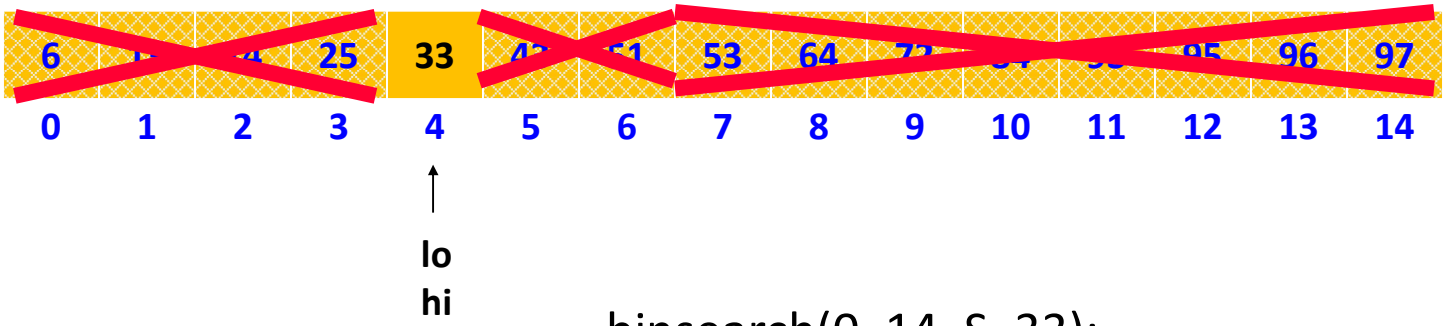
binsearch(4, 4, S, 33);

The section to be investigated is halved after each iteration

Example: Binary Search

```
int binsearch(int low, int high, int S[], int key)
{
   if (low <= high)
   {
       int mid = (low + high) / 2;
       if (S[mid]== key) return mid;
       else if (key < S[mid])
           return binsearch(low, mid-1, S, key);
       else
           return binsearch(mid+1, high, S, key);
    }
    else return -1;
}
```

key=33



binsearch(0, 14, S, 33);

binsearch(0, 6, S, 33);

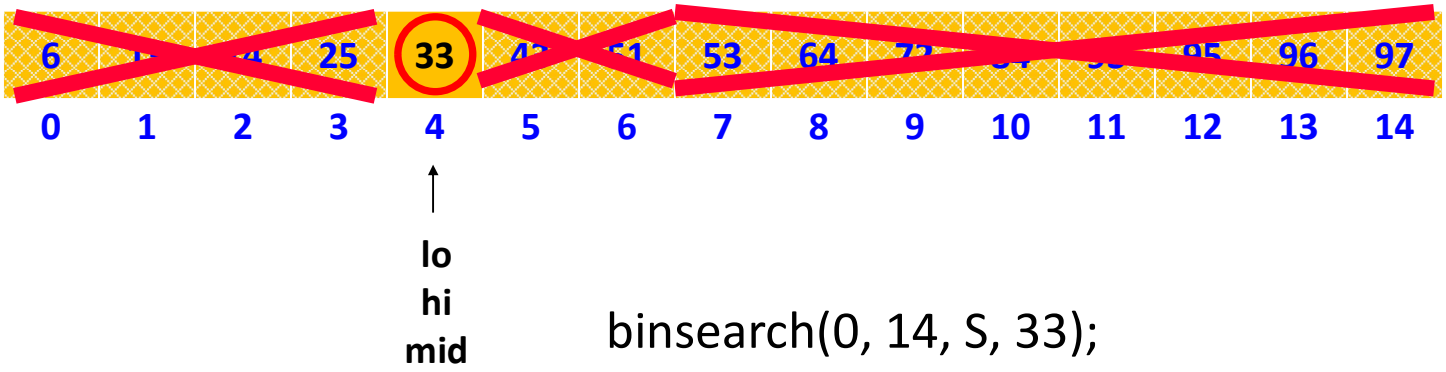
binsearch(4, 6, S, 33);

binsearch(4, 4, S, 33);

Example: Binary Search

```
int binsearch(int low, int high, int S[], int key)
{
    if (low <= high)
    {
        int mid = (low + high) / 2;
        if (S[mid]== key) return mid;
        else if (key < A[mid])
            return binsearch(low, mid-1, S, key);
        else
            return binsearch(mid+1, high, S, key);
     }
     else return -1;
}
```

key=33



binsearch(0, 14, S, 33);

binsearch(0, 6, S, 33);

binsearch(4, 6, S, 33);

binsearch(4, 4, S, 33);

Example 3: Palindrome

- **Definition.** ***Palindrome*** is a string that reads it from left to right is the same as reading it from right to left.

Example: NOON, DEED, RADAR, MADAM

Able was I ere I saw Elba

- To determine if a given string is palindrome:
  - Compare the first character and the last character of the string.
    - If they are equal: new string = old string that removes the first and last characters. Repeat comparison step.
    - If they are different: given string is not palindrome

Example 4: Palindrome: str[start….end]

- Base case : string has <=1 character (`start >= end`)

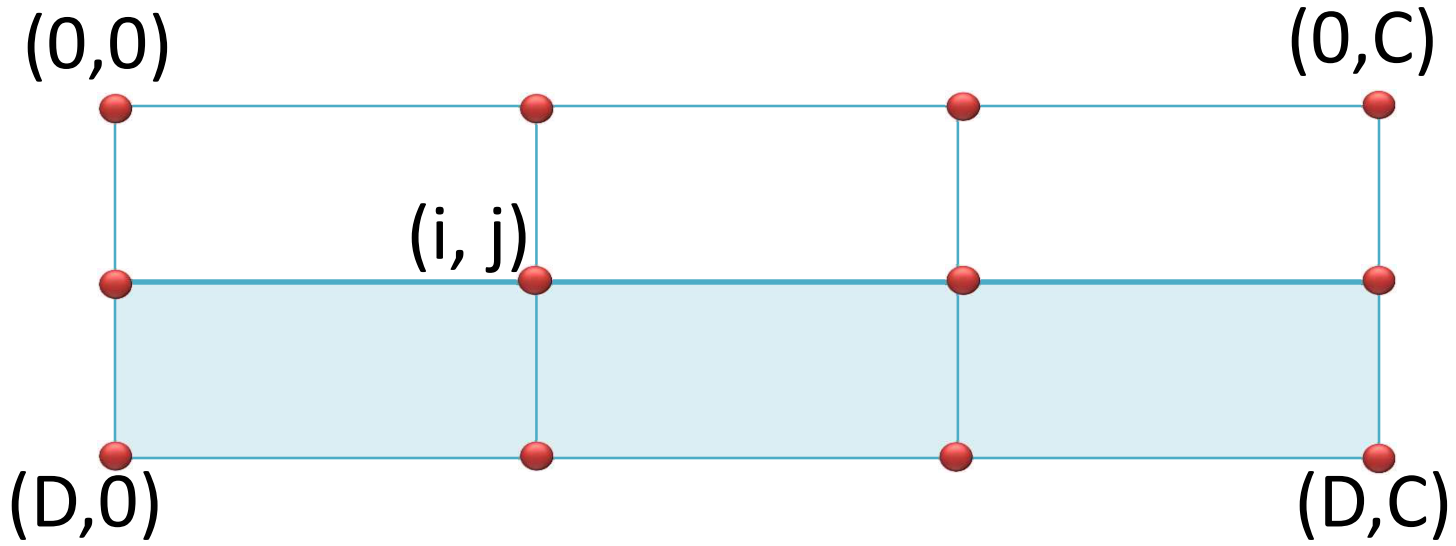
```
return true
```

- Recursive step:

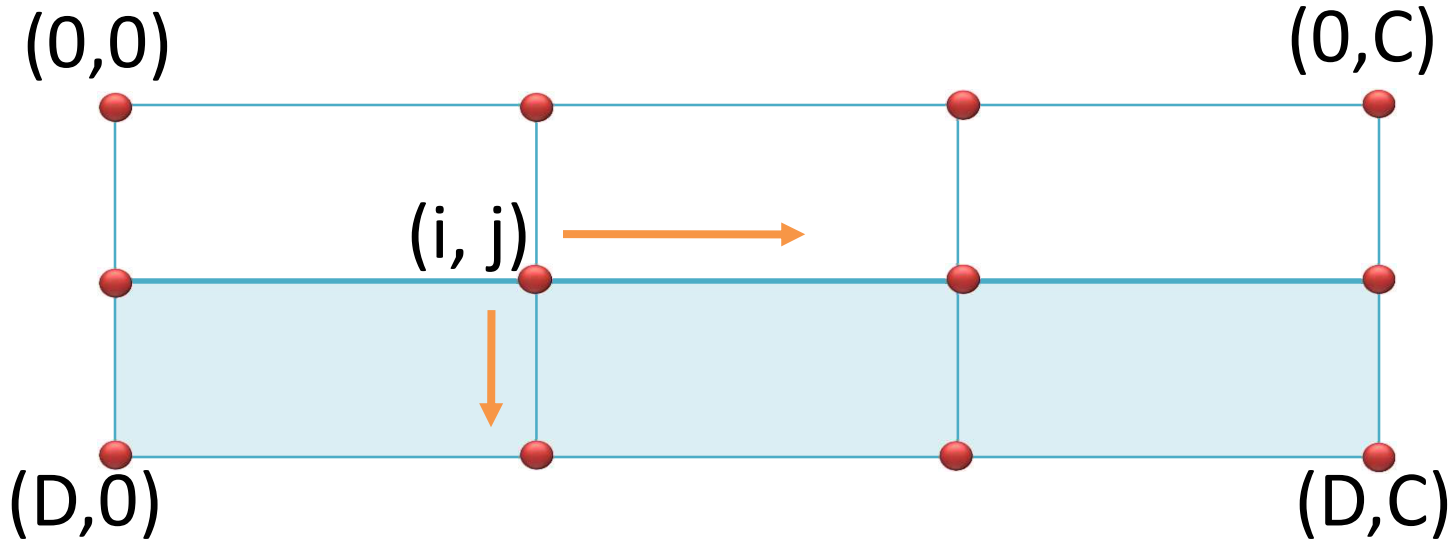
```
return true if (str[start]==str[end] &&
                palindrome(str, start+1, end-1))
```

Example 5: Path on grid

- Given the grid of size $D*C$.
- You are only allowed to move from one node to other node on the grid in either direction downwards or to the right.
- How many paths are there from node (i, j) to node (D, C).
  - Write function: `int CountPaths(int i, int j, int D, int C)`



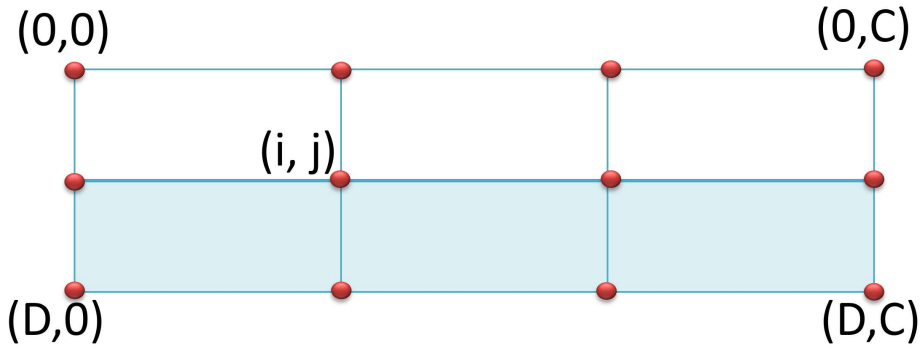
Example 5: Path on grid



- To solve the problem `CountPaths(i,j, D, C)`, we need to solve which subproblems ?
  - From node (i, j) there are only 2 ways:
    - Go down: to node $(i+1, j)$ `CountPaths(i+1,j, D, C)`
    - Go right: to node $(i, j+1)$ `CountPaths(i,j+1, D, C)`

➔`CountPaths(i,j, D, C) = CountPaths(i+1,j, D, C)`
`+ CountPaths(i,j+1, D, C)`

Basic case



- If you step over the edge of the grid (no longer on the grid), there is no path to the node (D, C):
  - `if (i > D) OR (j > C): CountPaths(i,j, D, C)` return 0

```
int CountPaths(int i, int j, int D, int C)
{
    if (i>D || j>C)  return 0;
    else
        return CountPaths(i + 1, j, D, C) +
               CountPaths(i, j + 1, D, C);
}
```

DONE ?

NO: because all subproblems will return 0

One more special case should be considered: when you are at line D or column C ➔ there is a path (this path does not need any steps)

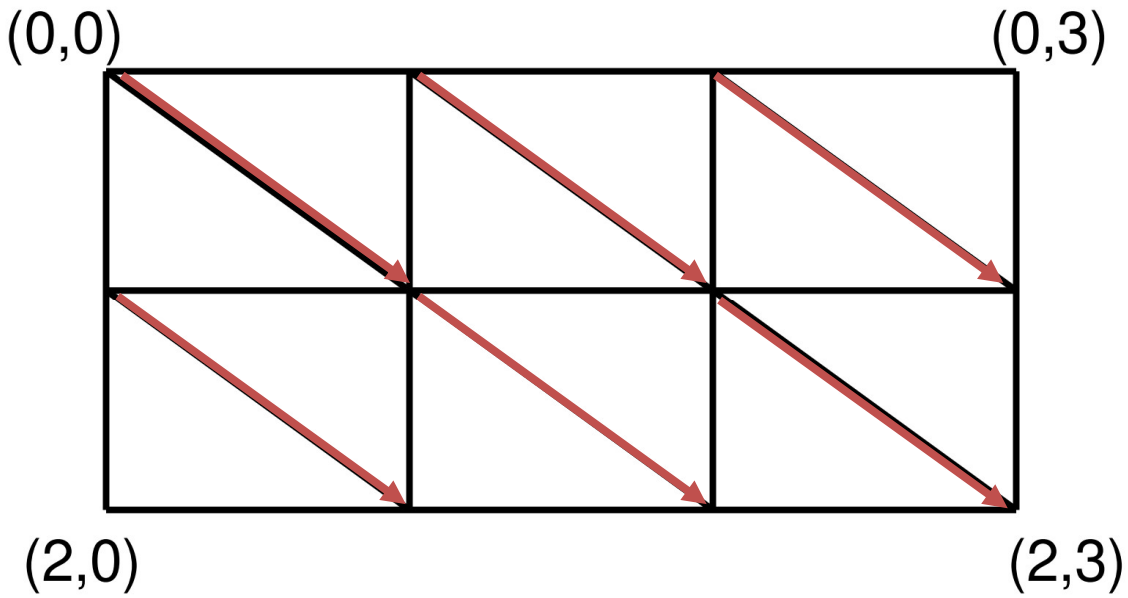
Example 5: Path on grid

- Given the grid of size $D*C$.
- You are only allowed to move from one node to other node on the grid in either direction downwards or to the right.
- How many paths are there from node (i, j) to node (D, C).
  - Function: `int CountPaths(int i, int j, int D, int C)`

```
int CountPaths(int i, int j, int D, int C)
{
    if (i>D || j>C)  return 0;
    else if(i==D || j == C)
        return 1;
    else
        return CountPaths(i + 1, j, D, C) +
               CountPaths(i, j + 1, D, C);
}
```

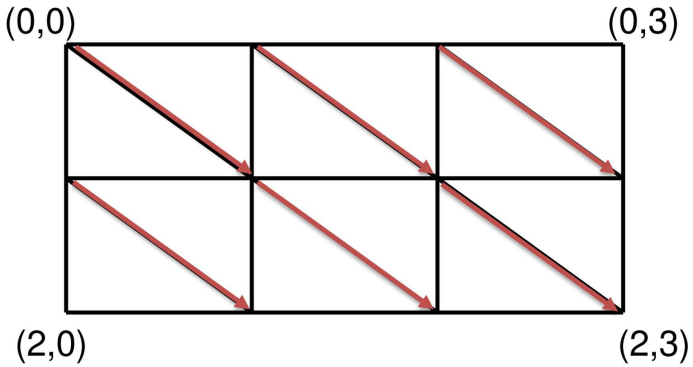
Example 5: Path on grid: VARIATION

- Besides going either downwards or to the right, we can also go diagonally.
- How many paths are there from node (i, j) to node (D, C).
  - Function: `int CountPaths(int i, int j, int D, int C)`



Example 5: Path on grid: VARIATION

```
int CountPaths(int i, int j, int D, int C)
{
    if (i>D || j>C)  return 0;
    else if(i==D || j == C)
        return 1;
    else
        return CountPaths(i + 1, j, D, C) +
               CountPaths(i, j + 1, D, C);
}
```




```
int CountPaths(int i, int j, int D, int C)
{
    if (i>D || j>C)  return 0;
    else if(i==D || j == C)
        return 1;
    else
        return CountPaths(i + 1, j, D, C) +
               CountPaths(i, j + 1, D, C) +
               CountPaths(i + 1, j + 1, R, C);
}
```

2. Recursion

2.1. The concept of recursion

2.2. Recursive algorithm diagram

2.3. Some illustrative examples

2.4. Analysis of recursive algorithm

2.5. Recursion and memorization

2.4. Analysis of recursive algorithm

- To analyze the computation time of recursive algorithm, we usually proceed as follows:
  - Let ***T(n)*** be the computation time of the algorithm
  - Build a recursive formula for ***T(n)***.
  - *Solve this recursive formula* to get the evaluation for ***T(n)***
- Generally, we only need a close assessment of the growth rate of $T(n)$, so ***solving the recursive formula*** for $T(n)$ is to evaluate the growth rate of $T(n)$ in the asymptotic notation

Example: Recursive Binary Search

```
int binsearch(int low, int high, int S[], int key)
{
    if (low <= high)
    {
        int mid = (low + high) / 2;
        if (S[mid]==key) return mid;
        else if (key < S[mid])
            return binsearch(low, mid-1, S, key);
        else
            return binsearch(mid+1, high, S, key);
     }
     else return -1;
}
```

➔ binsearch(0, n-1, S, key);

Let T(n): the computation time of binary search when when array S has n elements

$$T(n) = T(n/2) + \mathrm{K}$$
$$T(1) = c$$

where c and K are constants

Example 2: Recursive Binary Search

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

  a = 1

  b = 2

  d = 0

  Therefore, which condition applies?

  $1 = 2^0$, case 2 applies

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

Master Theorem

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

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

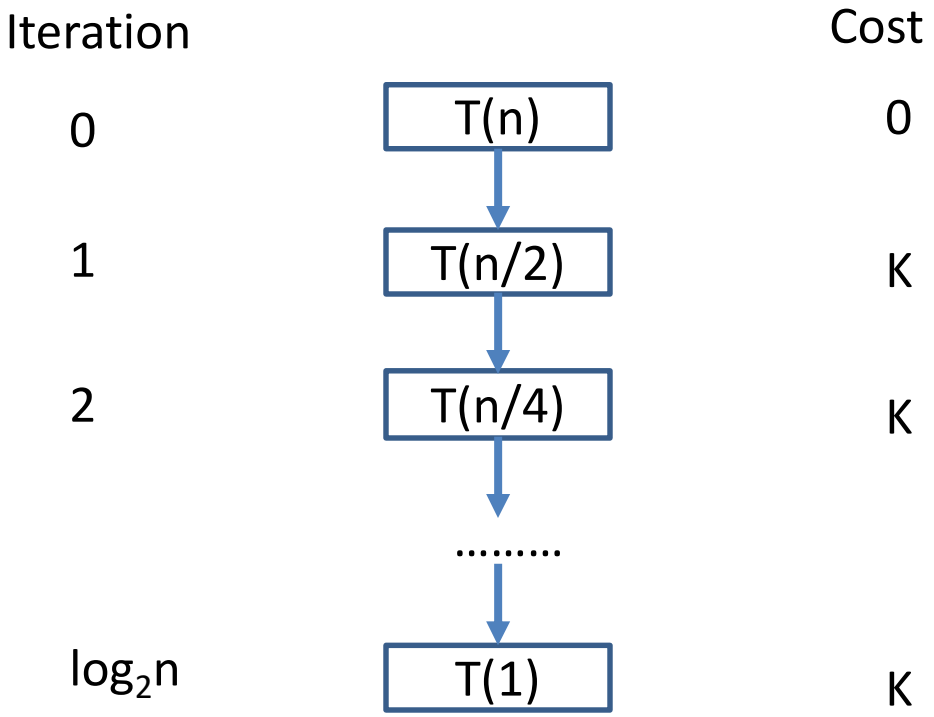
  $T(1) = c$

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

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

Solve recursive formula:

$$T(n) = T(n/2) + K, \quad T(1) = c$$



- Value of T(n) is equal to cost at all iterations:

$$T(n) = T(1) + \sum_{i=0}^{log_2 n}(K) = c + K\log_2 n$$

➔ So we have $\boldsymbol{T(n) = O(\log_2 n)}$

Solve recursive formula:

$$T(n) = T(n/2) + K, \quad T(1) = c$$

$$\begin{aligned} T(n) &= T(n/2) + K \\ &= T(n/4) + K + K \\ &= T(n/8) + K + K + K \\ &= \ldots. \\ &= T(n/2^i) + iK \\ &= T(1) + K \log_2 n \quad \text{where } i = \log_2 n \\ &= c + K \log_2 n \quad = \quad O(\log_2 n) \end{aligned}$$

➔ So we have $\boldsymbol{T(n) = O(\log_2 n)}$

2. Recursion

2.1. The concept of recursion

2.2. Recursive algorithm diagram

2.3. Some illustrative examples

2.4. Analysis of recursive algorithm

2.5. Recursion with memorization

2.5. Recursion with memorization

- In the previous section, we see that recursive algorithms for calculating Fibonacci numbers and calculating binomial coefficients were inefficient. To increase the efficiency of recursive algorithms without having to build iterative or recursive reduction procedures, we can use “recursion with memorization” technique.
- Using this technique, in many cases, we maintain the recursive structure of the algorithm and at the same time ensure its effectiveness. The biggest downside to this approach is the memory requirement.

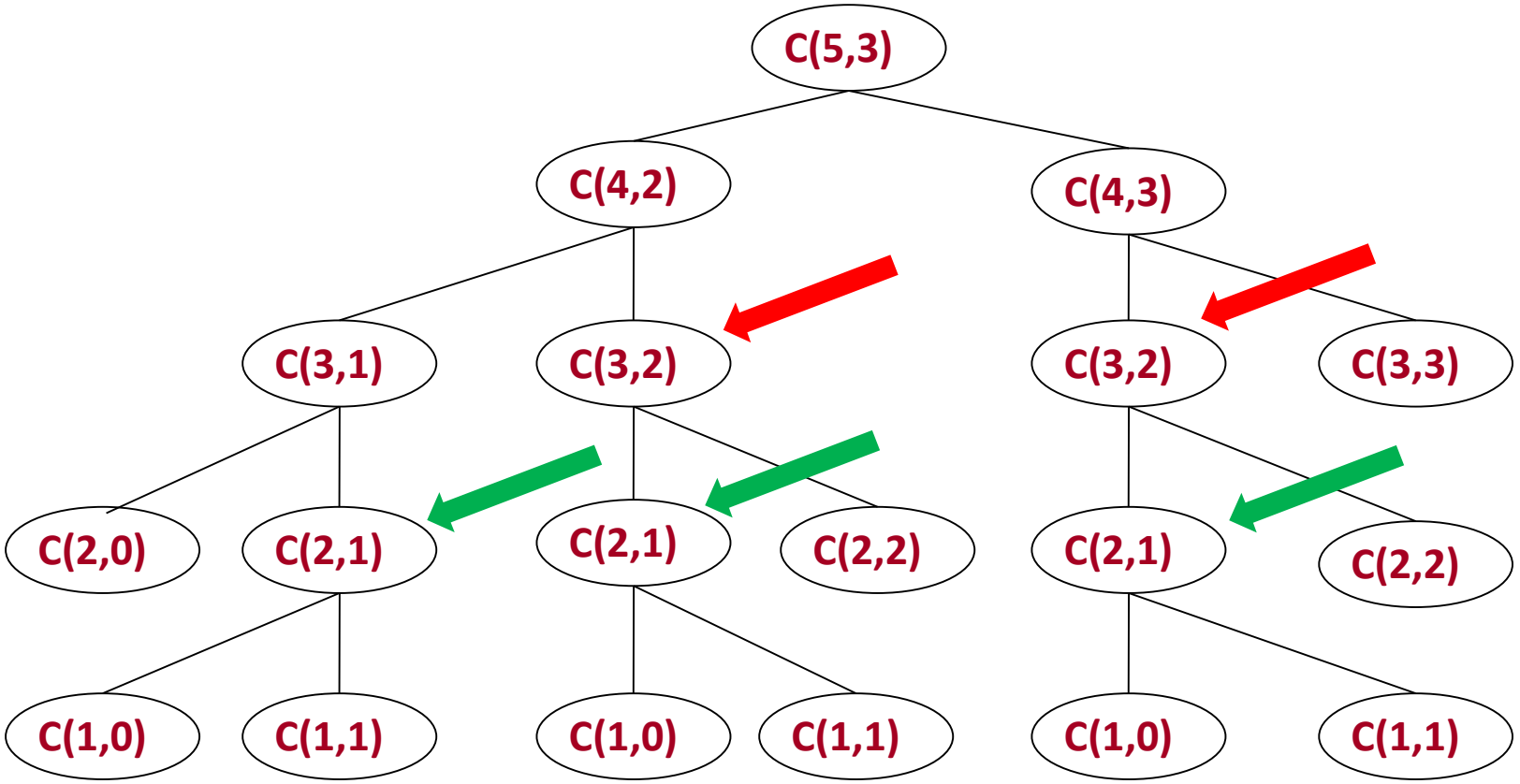
Duplication of subproblems

- Realizing that in recursive algorithms, whenever we need the solution of a subproblem, we must solve it recursively. Therefore, there are subproblems that are solved repeatedly. That leads to inefficiency of the algorithm. This phenomenon is called duplication of subproblem.

Example: Recursive algorithm to calculate C (5,3). The recursive tree that executes the call to function C (5,3) is shown in the following:

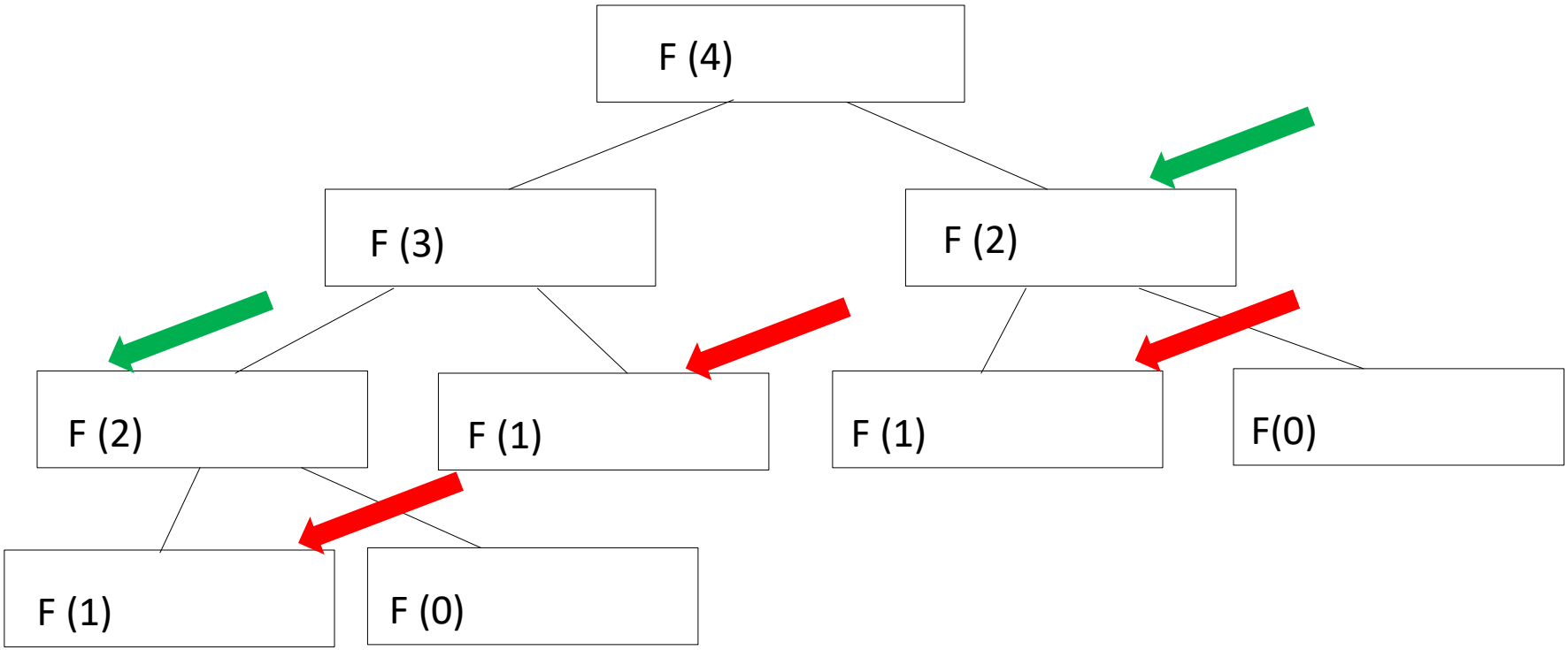
Example: Duplication of subproblems when calculating C(5,3)

```
int C(int n, int k){
    if ((k==0)||(k==n)) return 1;
    else return C(n-1,k-1)+C(n-1,k);
}
```



Example: Duplication of subproblems when calculating Fibonacci F(4)

```
int F (int n) {
   if  (n < 2) return n;
   else return F(n-1)+F(n-2);
}
```



Recursive with memorization

- To overcome this phenomenon, the idea of ***recursive with memorization*** is: We will use the variable to memorize information about the solution of subproblem right after the first time it is solved. This allows to shorten the computation time of the algorithm, because, whenever needed, it can be looked up without having to solve the subproblems that have been solved before.

Example: Recursive algorithm calculates binomial coefficients, we put a variable

- D[n][k] to record calculated value of C(n, k).
- Initially D[n][k]=0, when C(n, k) is calculated, this value will be stored in D[n][k]. Therefore, if D[n][k]>0 then it means there is no need to recursively call function C(n, k)

Example: Recursive with memorization to calculate C(n,k)

```
int C(int n,int k){
    if (D[n][k]>0)  return D[n][k];
    else{
            D[n][k] = C(n-1,k-1)+C(n-1,k);
            return D[n][k];
        }
}
```

Before calling function C(n, k), we need to initialize array D[ ][ ] as following:

- D[i][0] = 1, D[i][n]=1, where i = 0,1,..., n;
- D[i][j] = 0, for remaining values of i, j

```
int C(int n, int k){
    if ((k==0)||(k==n)) return 1;
    else return C(n-1,k-1)+C(n-1,k);
}
```

Contents

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

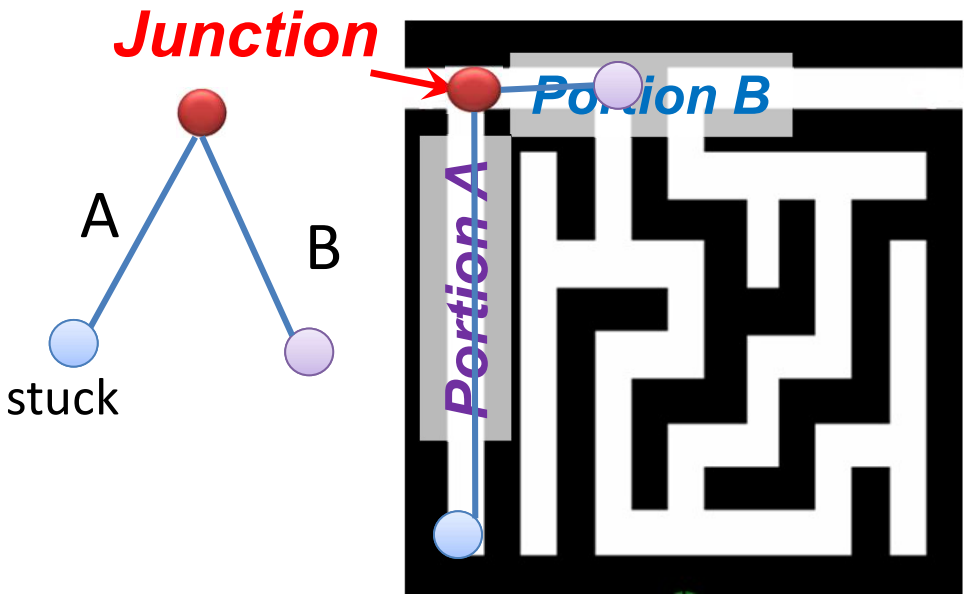
NGUYỄN KHÁNH PHƯƠNG
Bộ môn KHMT – ĐHBK HN

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**

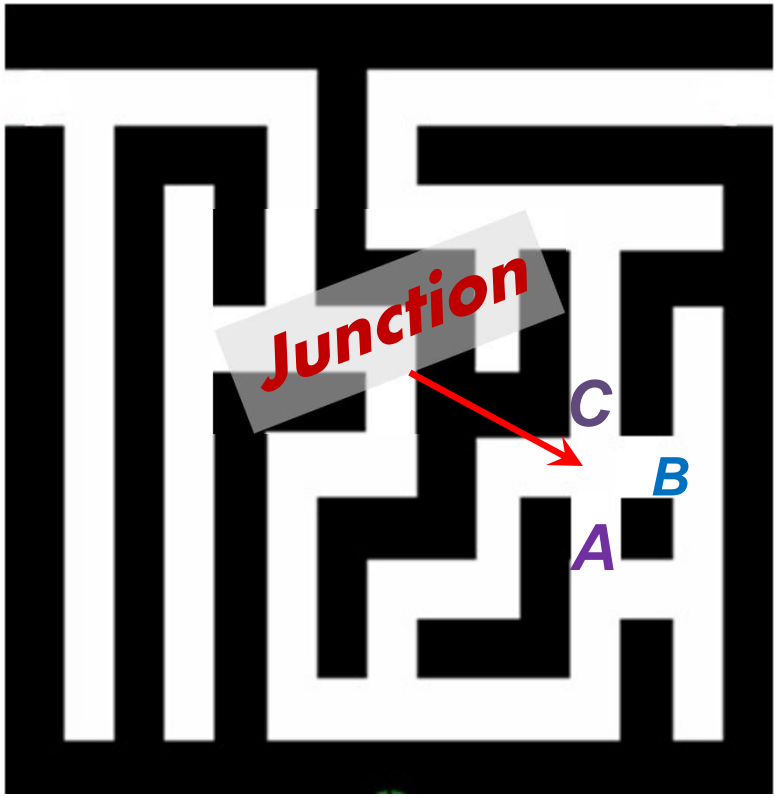


Backtracking idea

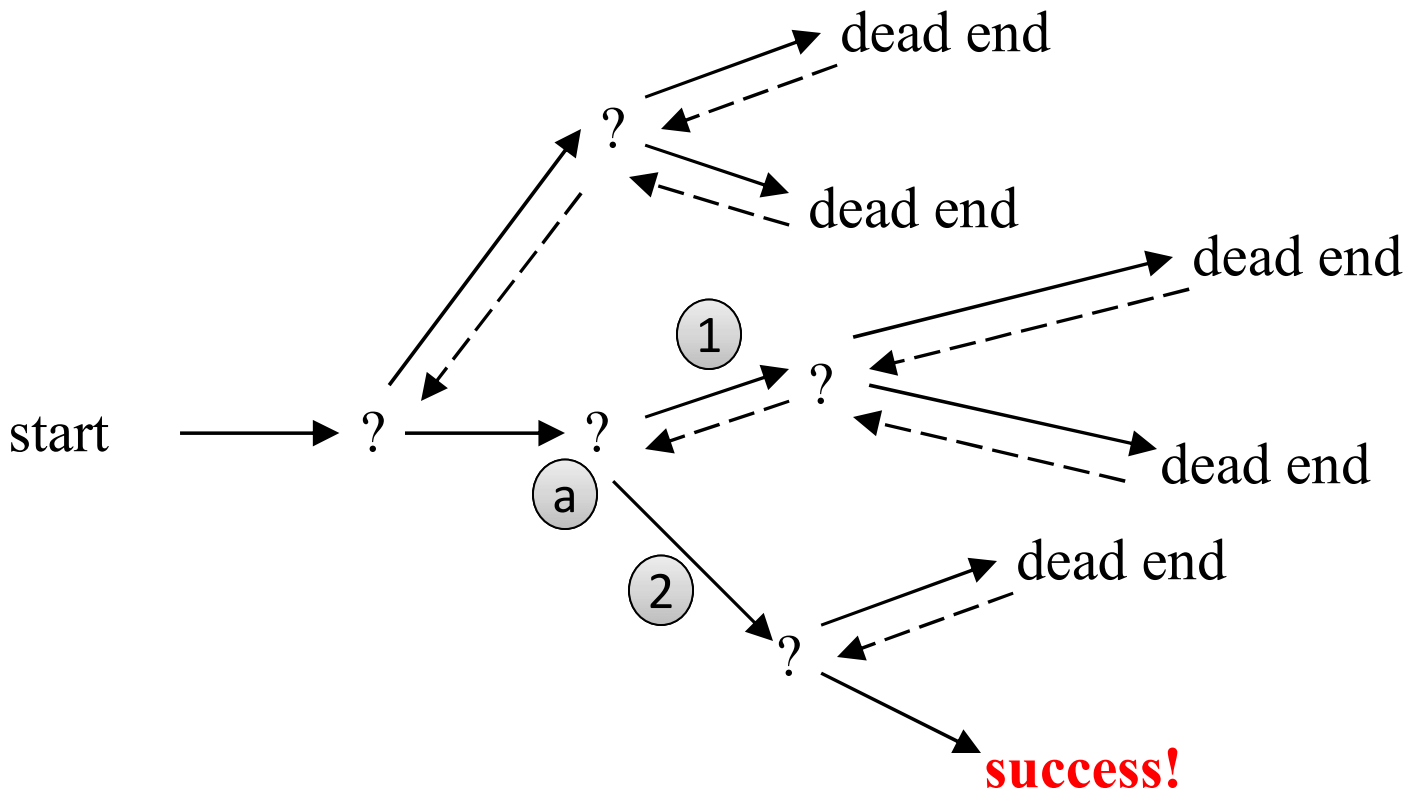
- Clearly, at a single junction you could have even more than 2 choices.

- 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:

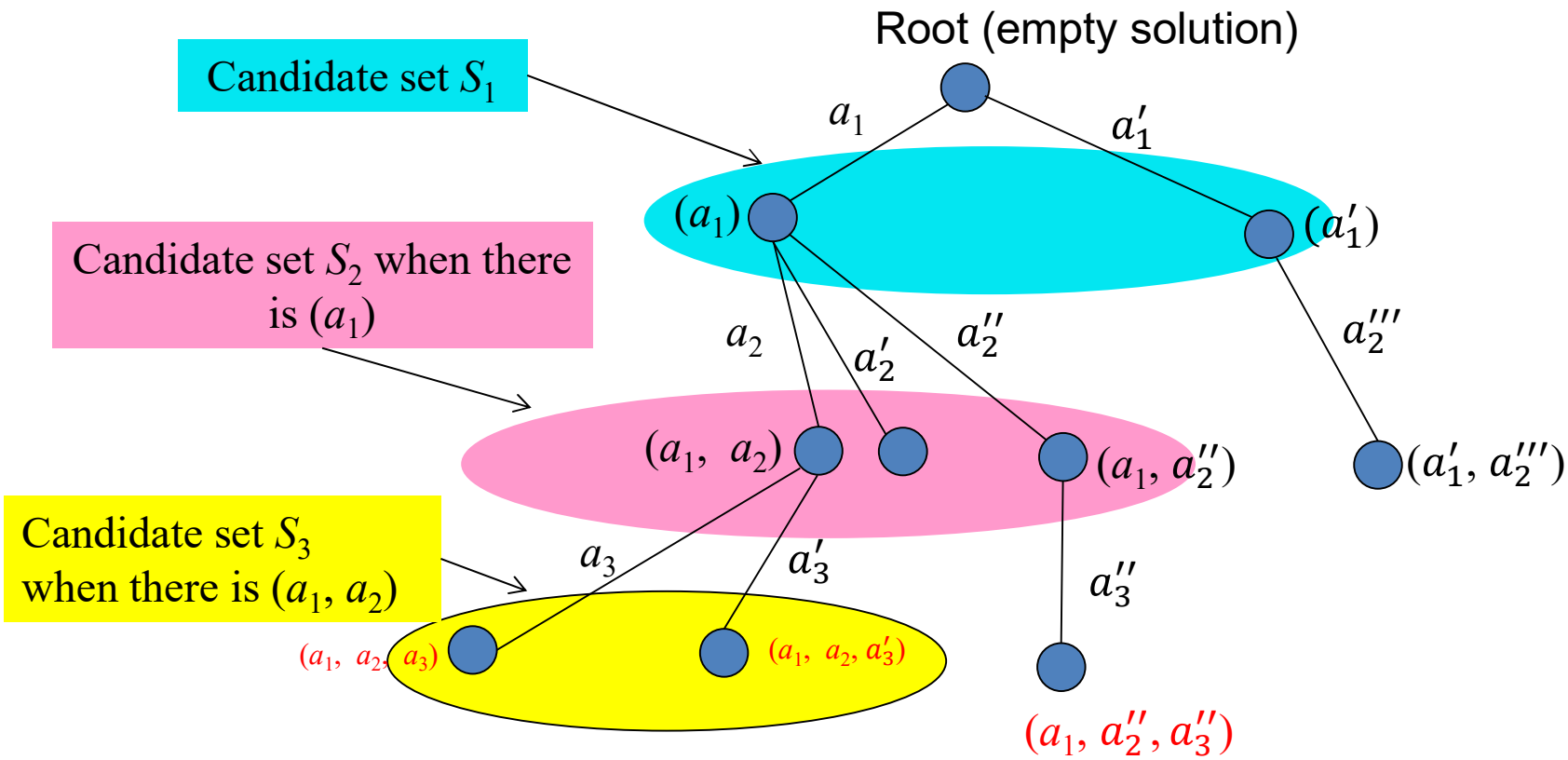
$$(a_1, a_2, ..., a_{k-1}, \mathbf{a_k})$$

  - 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)

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

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:

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

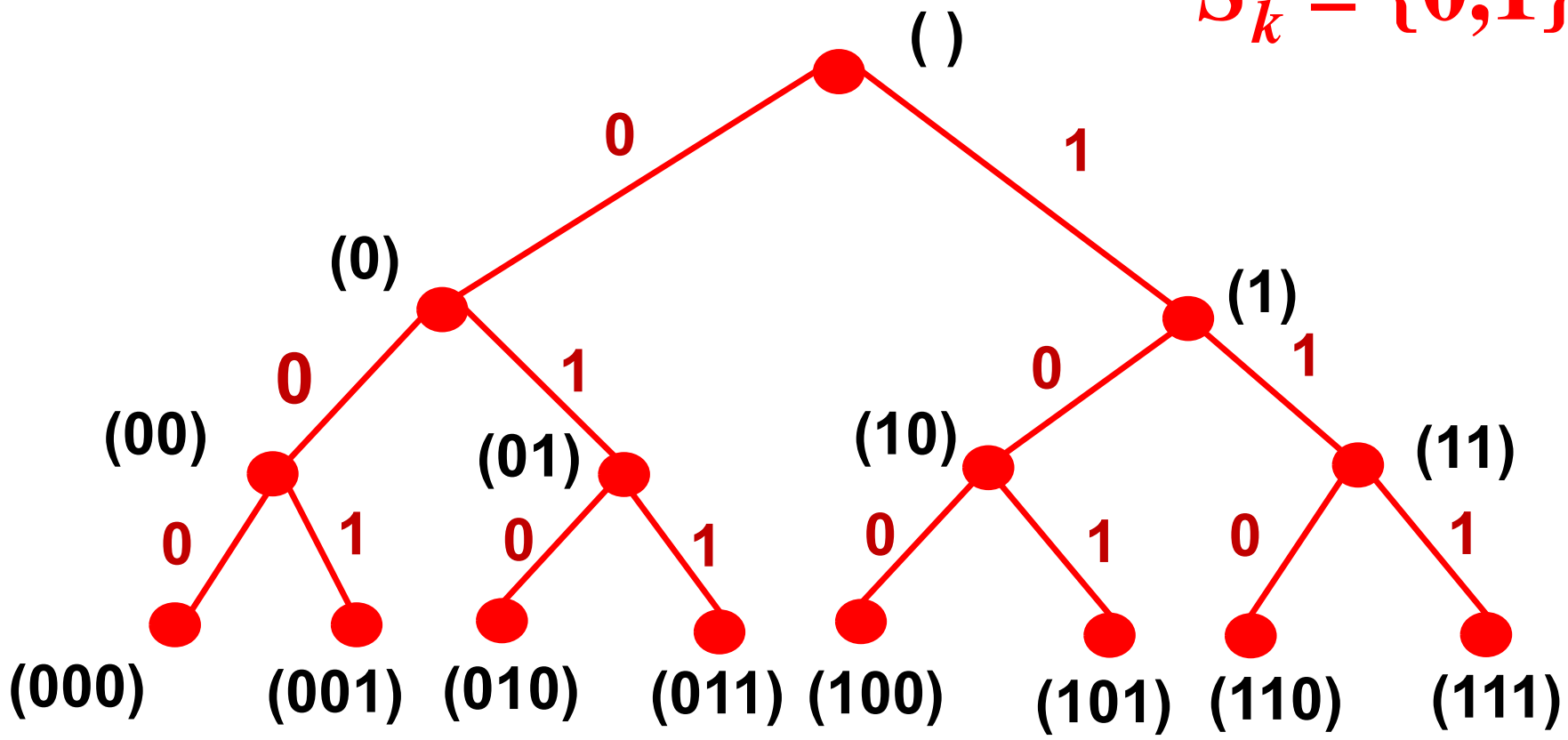
- We consider how to solve two issue keys to implement backtracking algorithm:
  - Build candidate set S_k: We have $S_1 = \{0, 1\}$. Assume we have binary string of length k-1 $(a_1, ..., a_{k-1})$, then $S_k = \{0,1\}$. Thus, the candidate sets for each position of the solution are determined.
  -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

```
for (y=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;
}
```

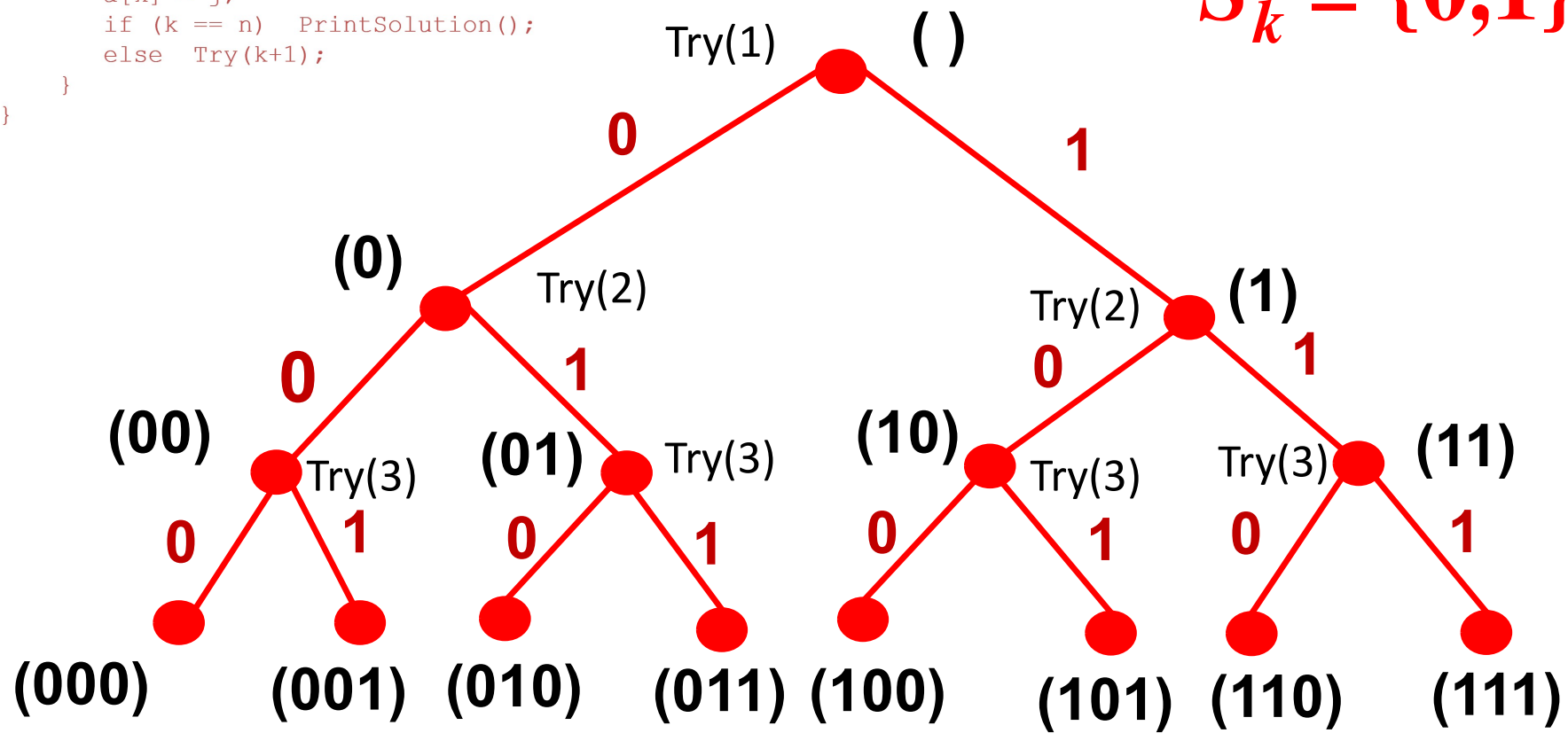
```
void  Try(int k)
{
   for (int j = 0; j<=1; j++)
   {
       a[k] = j;
       if (k == n)  PrintSolution();
       else  Try(k+1);
    }
}

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

NGUYỄN KHÁNH PHƯƠNG
Bộ môn KHMT – ĐHBK HN

Decision tree to enumerate binary strings of length 3

```
void  Try(int k)
{
   for (int j = 0; j<=1; j++)
   {
       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

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
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=a[k-1]+1;j<=n-m+k;j++)
```

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;
}
```

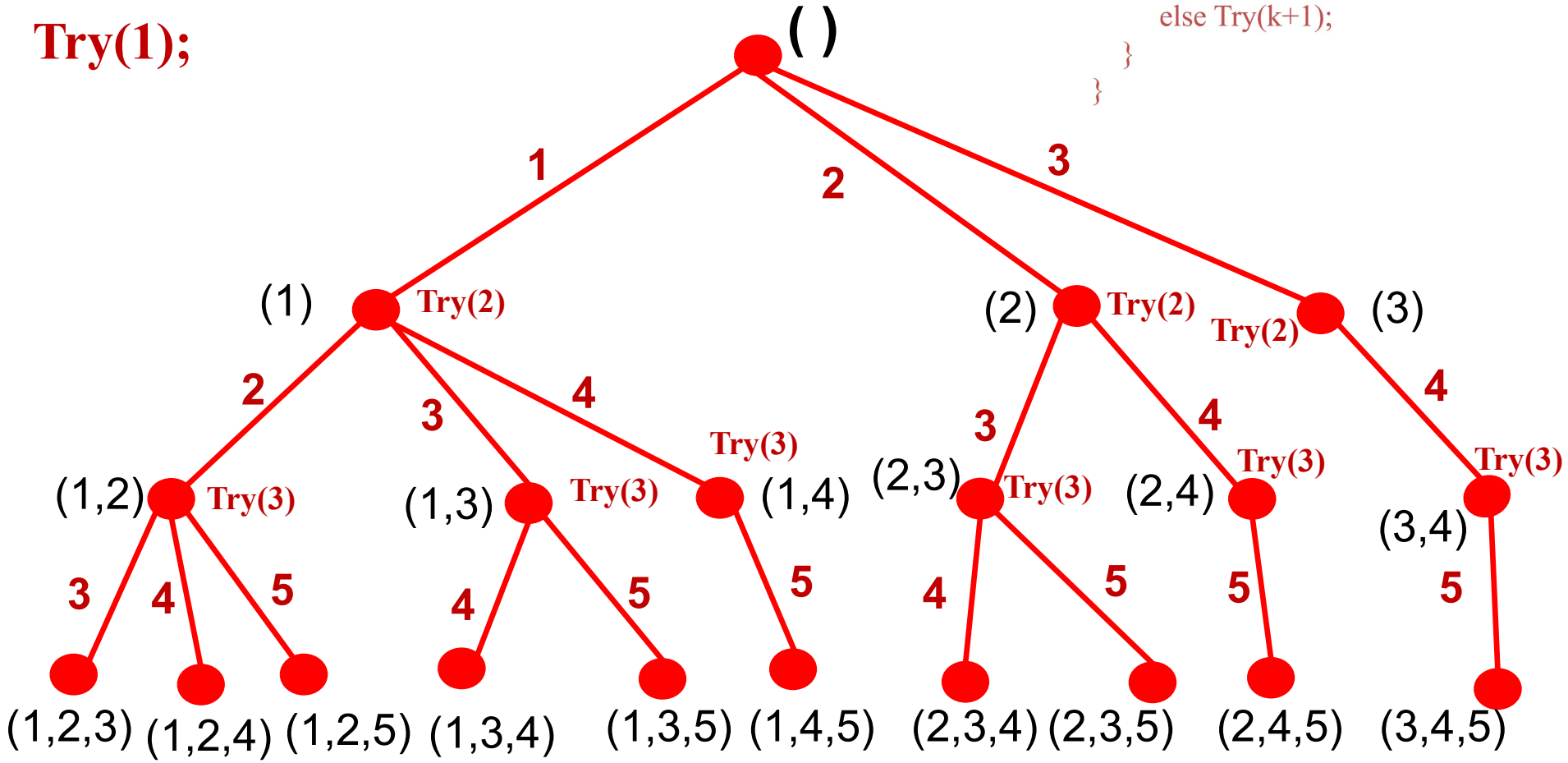
Program in C++ (Non Recursive)

```
#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;
}
```

Decision tree S(5,3)

```
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);
    }
}
```



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

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
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:

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

Example 3. Enumerate permutations

- Implement the loop to enumerate all elements of S_k:

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

Program in C++ (Recursive)

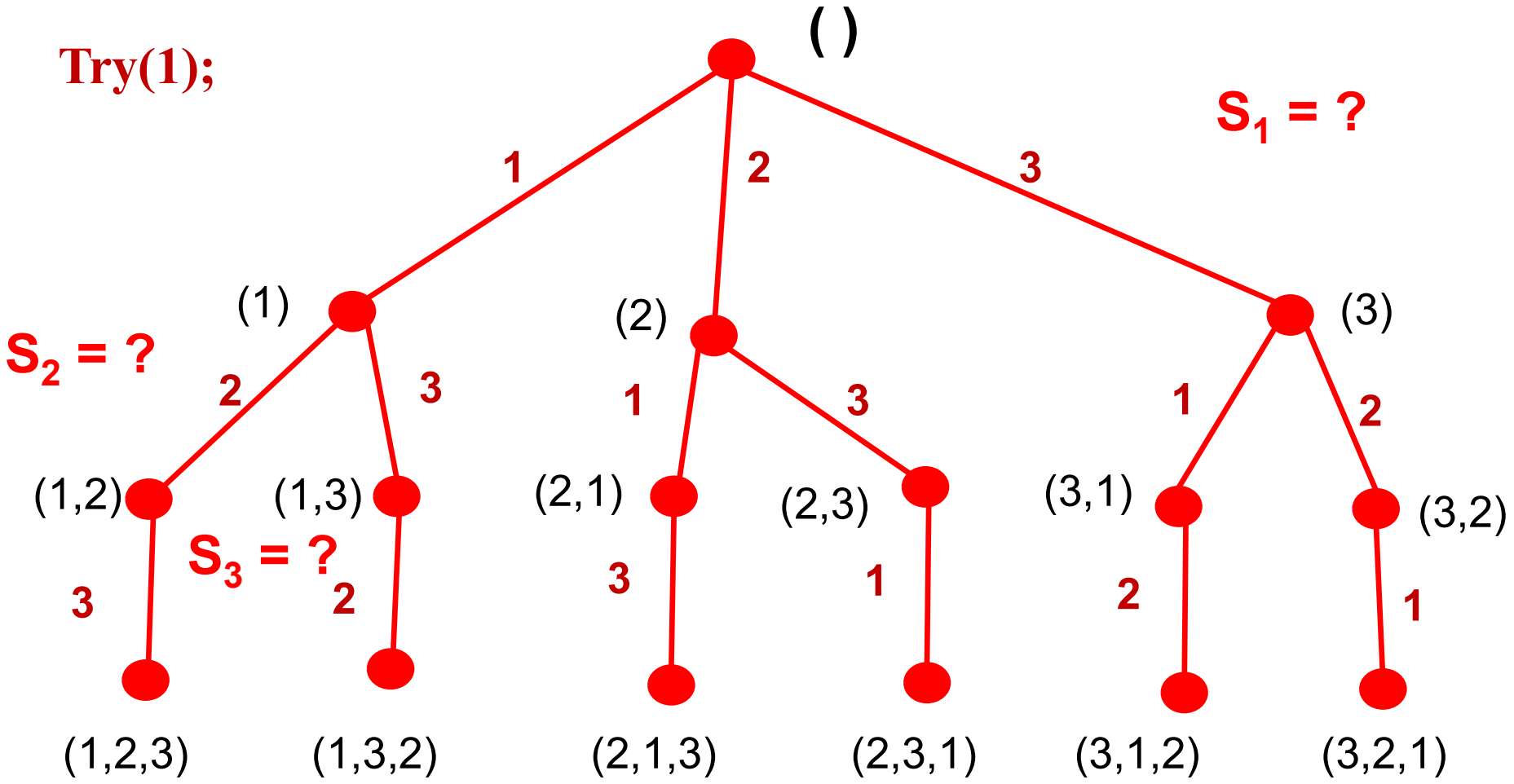
```
#include <iostream>
using namespace std;

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;
}
```

Decision tree to enumerate permutations of 1, 2, 3



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

Contents

4. Divide and conquer

4.1. Algorithm diagram

4.2. Some illustrative examples

4.1. Algorithm diagram

- Divide and Conquer (Chia để trị): consists of 3 operations:
  - Divide: Decompose the given problem S into some problems of same form but with smaller input size (called a subproblem) S_1, S_2, …
  - Conquer: Solve subproblem recursively
  - Combine: Synthesize the solutions of subproblems S_1, S_2,.. to obtain the solution of the original problem S.

- If the subproblem is small enough to be easily solved, then we solve it directly, otherwise: the subproblem is solved again by applying the above procedure recursively (i.e. dividing it again into smaller problem). Therefore, the divide and conquer algorithm is recursive algorithm => to analysis the complexity of the algorithm we can use recursive formula.

4.1. Algorithm diagram

To get a detailed description of the divide and conquer algorithm, we need to define:

- Critical size n_0 (problems with size less than n_0 don't need subdivision)
- Dimensions of each subproblem
- Number of subproblems
- Algorithm for synthesizing solutions of subproblems.

```
procedure D-and-C(n)
begin
    if (n ≤ n0) then
        Solve the problem directly
   else
   begin
          Divide the problem into K subproblems of size n/b
          for (each subproblem in K subproblems) do D-and-C(n/b)
          Synthesize the solutions of K subproblems to obtain the
          solution of the problem with size n.
   end;
end;
```

4.1. Algorithm diagram

```
procedure D-and-C(n)
begin
    if (n ≤ n0) then
        Solve the problem directly
   else
   begin
          Divide the problem into K subproblems of size n/b
          for (each subproblem in K subproblems) do D-and-C(n/b)
          Synthesize the solutions of K subproblems to obtain the
          solution of the problem with size n.
   end;
end;
```

Let T(n) be the computation time of the algorithm to solve the problem of size n

- D(n): time to divide
- C(n): time to synthesize the solutions

Then

$$T(n) = \begin{cases} c & when\ n \leq n_0 \\ kT\left(\frac{n}{b}\right) + D(n) + C(n) & when\ n > n_0 \end{cases}$$

where c is the constant

Example.

```
procedure D-and-C(int n)
begin
      if (n == 0)  return;
      D-and-C(n/2);
      D-and-C(n/2);
      for (int i =0 ; i < n; i++)
      begin
          Perform operations requires constant time
      end;
end;
```

What is the computation time of this procedure ?

4. Divide and conquer

4.1. Algorithm diagram

4.2. Some illustrative examples

- **Example 1. Binary search**
- Example 2. Multiply integer numbers

Example 1: Binary Search

Input: An array *S* consists of *n* elements: S[0],…,S[*n*-1] in ascending order; Value ***key*** with the same data type as array *S*.

Output: the index in array if ***key*** is found, -1 if ***key*** is not found

- **Binary search algorithm:** The value *key* either
  - **equals to the element at the middle of the array S,**
  - **or is at the left half (L) of the array S,**
  - **or is at the right half (R) of the array S.**

(The situation L (R) happen only when *key* is smaller (larger) than the element at the middle of the array S)

```
int binsearch(int low, int high, int S[], int key)
{
   if (low <= high)
   {
       int mid = (low + high) / 2;
       if (S[mid]==key) return mid;
       else if (key < S[mid])
           return binsearch(low, mid-1, S, key);
       else
           return binsearch(mid+1, high, S, key);
    }
    else return -1;
}
```

What is the computation time of binary search ?

4. Divide and conquer

4.1. Algorithm diagram

4.2. Some illustrative examples

- Example 1. Binary search
- **Example 2. Multiply integer numbers**

Example 2. Multiply 2 integer numbers

				9	8	1
			1	2	3	4
			3	9	2	4
		2	9	4	3	
	1	9	6	2		
	9	8	1			
1	2	1	0	5	5	4

- If each operand has n digits, then computation time is $\Theta(n^2)$
- Is there a better way?

Example 2. Multiply 2 integer numbers

- Problem: Given

$$x = x_{n-1}\, x_{n-2} \ldots x_1\, x_0 \text{ and}$$

$$y = y_{n-1}\, y_{n-2} \cdots y_1\, y_0$$

2 positive integer numbers with n digits. We need to calculate

$$z = z_{2n-1}\, z_{2n-2} \cdots z_1\, z_0$$

representing product of xy with $2n$ digits.

Example 2. Multiply 2 integer numbers - Karatsuba algorithm (1962)

- We have: $x = x_{n-1}\, x_{n-2} \ldots x_1\, x_0$ và $y = y_{n-1}\, y_{n-2} \ldots y_1\, y_0$

➔
$$x = x_{n-1} \times 10^{n-1} + x_{n-2} \times 10^{n-2} + \ldots + x_1 \times 10^1 + x_0 \times 10^0$$
$$y = y_{n-1} \times 10^{n-1} + y_{n-2} \times 10^{n-2} + \ldots + y_1 \times 10^1 + y_0 \times 10^0$$

- Thus: $z = z_{2n-1}\, z_{2n-2} \ldots z_1\, z_0 = x * y$

$$z = z_{2n-1} \times 10^{2n-1} + z_{2n-2} \times 10^{2n-2} + \ldots + z_1 \times 10^1 + z_0 \times 10^0$$
$$= (x_{n-1} \times 10^{n-1} + x_{n-2} \times 10^{n-2} + \ldots + x_1 \times 10^1 + x_0 \times 10^0) \times$$
$$\times (y_{n-1} \times 10^{n-1} + y_{n-2} \times 10^{n-2} + \ldots + y_1 \times 10^1 + y_0 \times 10^0)$$

Example 2. Multiply 2 integer numbers - Karatsuba algorithm (1962)

We have: $x = x_{n-1}\, x_{n-2} \ldots x_1\, x_0$ và $y = y_{n-1}\, y_{n-2} \ldots y_1\, y_0$

- Set:

$$a = x_{n-1}\, x_{n-2} \ldots x_{n/2+1}\, x_{n/2},$$
$$b = x_{n/2-1}\, x_{n/2-2} \ldots x_1\, x_0,$$
$$c = y_{n-1}\, y_{n-2} \ldots y_{n/2+1}\, y_{n/2},$$
$$d = y_{n/2-1}\, y_{n/2-2} \ldots y_1\, y_0.$$

Then:

$$x = a \times 10^{n/2} + b;$$
$$y = c \times 10^{n/2} + d,$$

- So:

$$z = x^*y = (a \times 10^{n/2} + b) \times (c \times 10^{n/2} + d)$$
$$= (a \times c)\, 10^n + (a \times d + b \times c)\, 10^{n/2} + b \times d.$$

➔To calculate a×c, a×d, b×c, b×d we must perform 4 multiplications of two *n*/2-digit numbers.

➔The given problem requires multiplication of two *n*-digit numbers (*x* and *y*): is transferred to problem of calculating 4 multiplications of two *n*/2-digit numbers

Example 2. Multiply 2 integer numbers - Karatsuba algorithm (1962)

- **Basic case:** The multiplication of two 1-digit integers can be done directly;
- **Divide:** if $n>1$ then the product of 2 integers with n digits can be represented through 4 products of 4 integer numbers with $n/2$ digits: $a*c$, $a*d$, $b*c$, $b*d$
- **Combine:** to calculate $z = xy$ when we already know the above 4 products, we only need to perform additions (can be done in $O(n)$) and multiply with power of 10 (can be done in $O(n)$, by inserting an appropriate number of digits 0 to the right).

$$x = a\times 10^{n/2} + b; \qquad z = x*y = (a\times 10^{n/2} + b) \times (c\times 10^{n/2} + d)$$

$$y = c\times 10^{n/2} + d, \qquad = (a\times c)\ 10^{n} + (a\times d + b\times c)\ 10^{n/2} + b\times d.$$

- Evaluate the computation time $T(n)$ of the algorithm:

$T(1) = 1$, $n = 1$

$T(n) = 4T(n/2) + n$, $n>1$

Using the master theorem: $T(n) = O(n^2)$

Is it possible to speed up??

Example 2. Multiply 2 integer numbers - Karatsuba algorithm (1962)

Karatsuba discovered how to perform 2 integer *n*-digit numbers multiplication requires only 3 multiplications of *n*/2-digit numbers as following:

- Set: $U = a \times c$, $V = b \times d$, $W = (a+b) \times (c+d)$

Then: $a \times d + b \times c = W - U - V$,

And calculate:

$$z = x*y = (a \times 10^{n/2} + b) \times (c \times 10^{n/2} + d)$$

$$= (a \times c)\ 10^{n} + (a \times d + b \times c)\ 10^{n/2} + b \times d$$

$$= U \times 10^{n} + (W-U-V) \times 10^{n/2} + V.$$

Karatsuba algorithm

```
function Karatsuba(x, y, n)
begin
   if n=1 then  return x[0]*y[0]
   else
   begin
        a:= x[n-1] ... x[n/2];
        b:= x[n/2 -1] ... x[0];
        c:= y[n-1] ... y[n/2];
        d:= y[n/2-1] ... y[0];
        U:= Karatsuba(a, c, n/2);
        V:= Karatsuba(b, d, n/2);
        W:= Karatsuba(a+b, c+d, n/2);
        return U*10^n + (W-U-V)*10^(n/2) + V;
   end;
end;
```

- Set: $U = a \times c$, $V = b \times d$, $W = (a+b) \times (c+d)$

Then: $a \times d + b \times c = W - U - V$,

And calculate:

$$z = x*y = (a \times 10^{n/2} + b) \times (c \times 10^{n/2} + d)$$
$$= (a \times c)\, 10^n + (a \times d + b \times c)\, 10^{n/2} + b \times d$$
$$= U \times 10^n + (W-U-V) \times 10^{n/2} + V.$$

Evaluate the computation time $T(n)$ of the algorithm:

$T(1) = 1$, $n = 1$

$T(n) = 3T(n/2) + cn$, $n > 1$

Using the master theorem: $T(n) = O(n^{log_2 3})$

As $\log_2 3 = 1.585 < 2$ ➔ already speed up

Contents

1. Brute force
2. Recursion
3. Backtracking
4. Divide and conquer
5. **Dynamic programming**

5. Dynamic programming

5.1. Algorithm diagram

5.2. Some illustrative examples

5.1. Algorithm diagram

The development of algorithm based on dynamic programming consists of 3 phases:

- **Decomposition:**
  - Divide the problem into smaller subproblems so that the smallest problem subproblem can be solved directly.
  - The original problem itself can be considered as the largest subproblem of this family of subproblems.
- **Store the solutions:**
  - Store the solutions of subproblems in a table. This is necessary because the solution of subproblems is often reused many times, and it improves the efficiency of the algorithm by not having to solve the same problem repeatedly.
- **Combine solutions:**
  - From the solution of smaller subproblems, in turn, seek to construct the solution of the problem of the larger size, until the solution of the original problem (which is the subproblem of the largest size) is obtained.

5.1. Dynamic programming Algorithm diagram

There are many similarities with the Divide and Conquer:

- **Divide and conquer:**
  - Divide the given problem into **independent** subproblems
  - Solve each subproblem (by recursion)
  - Combine solutions of subproblems into solutions of given problems.
- **Dynamic programming:**
  - Divide the given problem into **overlapping** subproblems
  - Solve each subproblem (by recursion)
  - Combine solutions of subproblems into solutions of given problem.
  - Each subproblem is only solved once by saving the solution in a table and when the subproblem is called again, just look up the solution in the table.

5.1. Dynamic programming Algorithm diagram

- The technique of solving subproblems of dynamic programming is a bottom-up process: small-sized subproblems are solved first, then use the solution of these subproblems to build the solution of the larger problem;
- Divide and conquer method: big problems are broken down into subproblems, and subproblems are treated recursively (top-down).

Dynamic programming diagram

1. Find dynamic progamming formula for problems based on subproblems
2. Implement dynamic programming formula: convert the formula into a recursive function
3. Storing the results of calculation functions

Comment: 1st step is difficult and most improtant. To perform 2nd and 3rd steps, we can apply the following general scheme:

```
map<problem, value> memory;

value DP(problem P) {
    if (is_base_case(P)) {
        return base_case_value(P);
    }

    if (memory.find(P) != memory.end()) {
        return memory[P];
    }

    value result = some value;
    for (problem Q in subproblems(P)) {
        result = combine(result, DP(Q));
    }

    memory[P] = result;
    return result;
}
```

Example: Fibonacci sequence

The first two numbers in the Fibonacci sequence are 1 and 1. The remaining numbers in the sequence are calculated by the sum of the two numbers immediately preceding it in the sequence.

Requirements: Find the n^{th} Fibonacci number.

Dynamic programming diagram

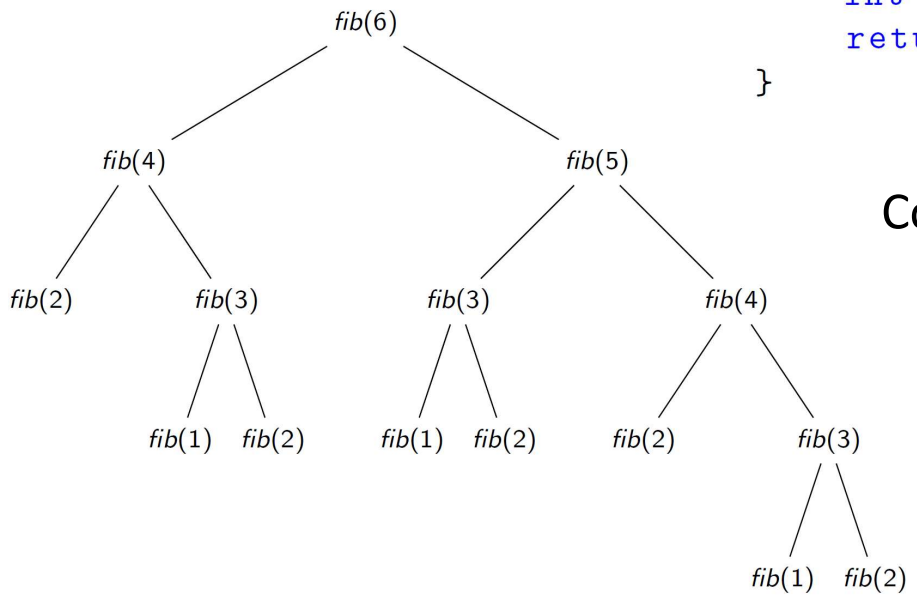
1. Find dynamic progamming formula for problems based on subproblems
2. Implement dynamic programming formula: convert the formula into a recursive function
3. Storing the results of calculation functions

1. Find dynamic programming formula

$fibonacci(1) = 1$
$fibonacci(2) = 1$
$fibonacci(n) = fibonacci(n-2) + fibonacci(n-1)$

2. Implement dynamic programming formula

```
int fibonacci(int n) {
    if (n <= 2) return 1;
    int res = fibonacci(n - 2) + fibonacci(n - 1);
    return res;
}
```



Complexity: exponenital ~ $O(2^n)$

Example: Fibonacci sequence

3. Store the results of calculation functions

```
map<int, int> mem;
int fibonacci(int n) {
    if (n <= 2) return 1;
    if (mem.find(n) != mem.end()) return mem[n];
    int res = fibonacci(n - 2) + fibonacci(n - 1);
    mem[n] = res;
    return res;
}
```

```
int mem[1000];
for (int i = 0; i < 1000; i++) mem[i] = -1;
int fibonacci(int n) {
    if (n <= 2) return 1;
    if (mem[n] != -1) return mem[n];
    int res = fibonacci(n - 2) + fibonacci(n - 1);
    mem[n] = res;
    return res;
}
```

Example: Fibonacci sequence

```
int mem[1000];
for (int i = 0; i < 1000; i++) mem[i] = -1;
int fibonacci(int n) {
    if (n <= 2) return 1;
    if (mem[n] != -1) return mem[n];
    int res = fibonacci(n - 2) + fibonacci(n - 1);
    mem[n] = res;
    return res;
}
```

What is the complexity?

When call `Fibonacci(n)` : there are n possibilities for input to the function: 1, 2, …, n

For each input:

- Either the result is taken directly from table if it has been calculated before: $O(1)$ if assume taking a value from memory just requires $O(1)$
- Or need to calculate the result and then store it: $O(1)$ if assume each recursive call and summing only take a constant amount of time

Each input is called up to 1 time

Total time is $O(n)$

5. Dynamic programming

5.1. Algorithm diagram

5.2. Some illustrative examples

- **Example 1. Maximum subarray problem**
- Example 2. Longest common subsequence
- Example 3. Travelling salesman problem

Example 1: The maximum subarray problem

- Given an array of n numbers:

  $a_1, a_2, \ldots, a_n$

  The contiguous subarray $a_i, a_{i+1}, \ldots, a_j$ with $1 \le i \le j \le n$ is a subarray of the given array and $\sum_{k=i}^{j} a_k$ *is called as the value of this subarray*

  The task is to find the maximum value of all possible subarrays, *in other words, find the maximum* $\sum_{k=i}^{j} a_k$. The subarray with the maximum value is called as the maximum subarray.

Example: Given the array -2, **11, -4, 13**, -5, 2 then the maximum subarray is 11, -4, 13 with the value = 11+ (-4)+13 =20

Example 1: The maximum subarray problem

1.1.1. Brute force

$$\frac{n^3}{6}+\frac{n^2}{2}+\frac{n}{3}$$

1.1.2. Brute force with better implement

$$\frac{n^2}{2}+\frac{n}{2}$$

1.1.3 Recursive algorithm

$n \log n$

1.1.4. Dynamic programming

n

Dynamic programming for maximum subarray problem

Dynamic programming diagram

1. Find dynamic progamming formula for problems based on subproblems
2. Implement dynamic programming formula: convert the formula into a recursive function
3. Storing the results of calculation functions

Step 1. Find dynamic programming formula

- Let `max_sum(i)` the value of maximum subarray of array $a_1, a_2, ..., a_i$, $i = 1, .., n$.
- Basic case: `max_sum(1) = max(0, a[1])`
- Find recursive formula for `max_sum(i) = ?`
  - Relate to `max_sum(i-1) ?`
  - Is it possible to construct solution of problem size i from the solutions of problems size less than i ?

Let `max_sum(i)` be the value of the maximum subarray of array $a_1, a_2, ..., a_i$, **and this subarray ends at a_i (this subarray includes a_i)**

Dynamic programming for maximum subarray problem

Dynamic programming diagram

1. Find dynamic progamming formula for problems based on subproblems
2. Implement dynamic programming formula: convert the formula into a recursive function
3. Storing the results of calculation functions

Step 1. Find dynamic programming formula

- Let `max_sum(i)` the value of maximum subarray of array a_1, a_2, ..., a_i , $i = 1, .., n$ and this subarray ends at a_i (this subarray includes a_i)
- Basic case: `max_sum(1) = a[1]`
- Find recursive formula for `max_sum(i)`

```
max_sum(i)= max(a[i], a[i] + max_sum(i-1))
```

➔ Solution to problem: $\max_{1 \le i \le n} \max\_sum(i)$

Dynamic programming for maximum subarray problem

Dynamic programming diagram

1. Find dynamic progamming formula for problems based on subproblems
2. Implement dynamic programming formula: convert the formula into a recursive function
3. Storing the results of calculation functions

Step 2. Implement dynamic programming formula

Basic case: `max_sum(1) = a[1]`

Recursive: `max_sum(i)= max(a[i], a[i] + max_sum(i-1))`

```
int a[1000];
int max_sum(int i) {
  if (i == 1) return a[i];
  int res = max(a[i],a[i] + max_sum(i - 1));
  return res;
}
```

Dynamic programming for maximum subarray problem

Dynamic programming diagram

1. Find dynamic progamming formula for problems based on subproblems
2. Implement dynamic programming formula: convert the formula into a recursive function
3. Storing the results of calculation functions

Step 3. Storing the results of calculation functions

```
int a[1000];
int mem[1000];
bool comp[1000];
memset(comp, 0, sizeof(comp));

int max_sum(int i) {
    if (i == 1) return a[i];
    if (comp[i]) return mem[i];
    int res = max(a[i], a[i] + max_sum(i - 1));
    mem[i] = res;
    comp[i] = true;
    return res;
}
```

The maximum subarray problem

In the main function, we need to call `max_sum(n);` this function will calculate all values `max_sum(i)` for all $1 \leq i \leq n$

Then, solution to the problem is the maximum of the `max_sum(i)` values already stored in array `mem[i]`

```
int maximum = 0;
for (int i = 1; i<= n; i++) {
    maximum = max(maximum, mem[i]);
}
```

```
maximum = *max_element(mem+1, mem+n+1);
```

```
printf("%d\n", maximum);
```

Dynamic programming for maximum subarray problem

```
int a[1000];
int mem[1000];
bool comp[1000];
memset(comp, 0, sizeof(comp));

int max_sum(int i) {
    if (i == 1) return a[i];
    if (comp[i]) return mem[i];
    int res = max(a[i], a[i] + max_sum(i - 1));
    mem[i] = res;
    comp[i] = true;
    return res;
}
```

Complexity ?

When call `max_sum(`n`)` : there are n possibilities input for function: 1, 2, …, n

For each input:

- Either the result is taken directly from table if it has been calculated before: $O(1)$ if assume taking a value from memory just requires $O(1)$
- Or need to calculate the result and then store it: $O(1)$ if assume each recursive call and function max only take a constant amount of time

Each input is called up to 1 time

Total time is $O(n)$

The max subarray problem: Trace

```
int a[1000];
int mem[1000];
bool comp[1000];
memset(comp, 0, sizeof(comp));

int max_sum(int i) {
    if (i == 1) return a[i];
    if (comp[i]) return mem[i];
    int res = max(a[i], a[i] + max_sum(i - 1));
    mem[i] = res;
    comp[i] = true;
    return res;
}
```

How to know max subarray consists which elements?

- Method 1: Trace using recursion

```
void trace(int i) {
    if (i != 1 &&
            mem[i] == a[i] + mem[i-1])
        trace(i - 1);
    printf("%d ", a[i]);
}
```

- Method 2: Trace using loop

```
int maximum = 0, pos = -1;
for (int i = 1; i <= n; i++) {
    maximum = max(maximum, mem[i]);
    if (maximum == mem[i]) pos = i;
}

printf("%d\n", maximum);
int L = pos, R = pos, sum = a[L];
while (sum != maximum){
    --L;
    sum += a[L];
}
printf("%d %d", L, R);
```

5. Dynamic programming

5.1. Algorithm diagram

5.2. Some illustrative examples

- Example 1. Maximum subarray problem
- **Example 2. Longest common subsequence**
- Example 3. Travelling salesman problem

Longest common subsequence (LCS)

Given the sequence of n elements $A = <a_0, a_1, ..., a_{n-1}>$

Definition of subsequence: if you delete 0 element or some elements of sequence A then you will get a sequence of A.

- Example: A = [5, 1, 8, 1, 9, 2]
  - Some subsequence of A:
    - [5, 1, 8, 1, 9, 2]
    - [5, 8, 9]
    - [1, 1]
    - []
  - Not subsequence of A:
    - [2, 5]
    - [10]

Given 2 sequences A and B, we say sequence Z is *common subsequence* of A and B if Z is subsequence of both A and B.

LCS problem: Given 2 sequences

$$A = <a_0, a_1, \ldots, a_{m-1}>$$

$$B = <b_0, b_1, \ldots, b_{n-1}>$$

Find the longest common subsequences A and B.

Applications: In biology, when examining genes

Longest common subsequence (LCS)

Example:

A = <K, J, **C**, **D**, E, **F**, G>

B = <**C**, C, E, **D**, E, G, **F**>

- Sequence Z = <C, D, F> is a common subsequence.
- Sequence <B, F, G> is not common subsequence.
- Sequence <C, D, E, G> is the longest common subsequence as we can not find and other common subsequence with length >=5.

Longest common subsequence (LCS)

LCS problem: Given 2 sequences

$A = <a_0, a_1, \ldots, a_{m-1}>$,

$B = <b_0, b_1, \ldots, b_{n-1}>$.

Find the longest common subsequence of A and B.

- **Direct approach:**
  - Browse all subquences of A ➔ *number of subsequences* $= 2^m$
  - For each subsequence of A, we check if it is subsequence of B. Complexity: $O(n)$

  ➔Complexity of direct apporach: $O(n.2^m)$

- **Dynamic programming:**

Step 1. Find dynamic programming formula

  - Let `lcs(i,j)` the length of common subsequence of $A_i = <a_0, a_1, \ldots, a_i>$ and $B_j <b_0, b_1, \ldots, b_j>$ where $-1 \le i \le m-1$ và $-1 \le j \le n-1$
  - Basic case: `lsc(i,-1)` $= 0$ và `lsc(-1,j)` $= 0$
  - Recursive function to calculate `lcs(i,j)`?

Longest common subsequence (LCS): Dynamic programming

- Clearly

  `lcs(i, -1) = 0,` $i = -1, 0, 1, ..., m-1$

  `lcs(-1, j) = 0,` $j = -1, 0, 1, ..., n-1.$

- Assume $i \geq 0, j \geq 0$, we need to calculate $lcs(i, j)$ which is length of LCS of two sequences $A_i = <a_0, a_1, ..., a_i>$ and $B_j <b_0, b_1, ..., b_j>$. There are 2 possibilities:
  - If $a_i = b_j$:
    - LCS of A_i and B_j can obtain by including a_i into LCS of 2 sequences A_{i-1} and B_{j-1} ➔ `lcs(i, j) = lcs (i-1, j-1) + 1`
  - If $a_i \neq b_j$:
    - LCS of A_i and B_j is the longest subsequence among 2 LCS of (A_i and B_{j-1}) and of (A_{i-1} và B_j) ➔ `lcs(i, j) = max {lcs(i, j-1), lcs (i-1, j)}`
- Therefore, we get the formula to calculate `lcs(i, j)`:

$$lcs(i,j) = \begin{cases} 0, & \text{if } i = -1 \text{ or } j = -1, \\ lcs(i-1, j-1)+1, & \text{if } i, j \geq 0 \text{ and } a_i = b_j \\ \max\{lcs(i, j-1), lcs(i-1, j)\}, & \text{if } i, j \geq 0 \text{ and } a_i \neq b_j. \end{cases}$$

Dynamic programming: Recursive implementation

$$lcs(i,j) = \begin{cases} 0, & \text{if } i = -1 \text{ or } j = -1, \\ lcs(i-1, j-1) + 1, & \text{if } i, j \geq 0 \text{ and } a_i = b_j \\ \max\{lcs(i, j-1), lcs(i-1, j)\}, & \text{if } i, j \geq 0 \text{ and } a_i \neq b_j. \end{cases}$$

```
string a = "bananinn", b = "kaninan";
int mem [1000][1000];
memset (mem , -1 , sizeof (mem ));

int lcs( int i , int j ) {
    if (i == -1 || j == -1) return 0;
    if ( mem[i][j ] != -1) return mem[i][j];

    int res;
    if (a[i ] == b[j ])
        res =  1 + lcs(i - 1 , j - 1);
    else
    {
        res = max(lcs (i , j - 1), lcs (i - 1 , j));
    }
    mem [i][j ] = res;
    return res;
}
```

$a =$ "bananinn"

$b =$ "kaninan"

LCS = “aninn”

Dynamic programming: Recursive implementation

In the main function, we only need to call

```
lcs(a.length()-1, b.length()-1);
```

```
int main()
{
    int m = a.length();
    int n = b.length();
    int answer = lcs(m-1,n-1);
    printf ("%d \n", answer);
    return 0;
}
```

LCS: Dynamic programming

```
string a = "bananinn", b = "kaninan";
int mem [1000][1000];
memset (mem , -1 , sizeof (mem ));

int lcs( int i , int j ) {
   if (i == -1 || j == -1) return 0;
   if ( mem[i][j ] != -1) return mem[i][j];

   int res;
   if (a[i ] == b[j ])
       res =  1 + lcs(i - 1 , j - 1);
   else
   {
       res = max(lcs (i , j - 1), lcs (i - 1 , j));
   }
   mem [i][j ] = res;
   return res;
}
```

Complexity ?

When call `lcs(len(a)-1, len(b)-1)`: there are $m*n$ input possibilities for function

For each input:

- Either the result is taken directly from table if it has been calculated before: $O(1)$ if assume taking a value from memory just requires $O(1)$
- Or need to calculate the result and then store it: $O(1)$ if assume each recursive call and function max, summing only take a constant amount of time

Each input is called up to 1 time

Total time is $O(m*n)$