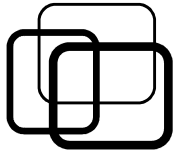
Recursion

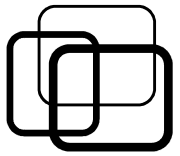
Inst. Nguyễn Minh Huy

Contents



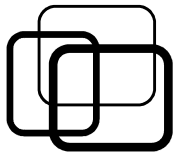
- Introduction.
- Classifications.
- Applications.

Contents



- **Introduction.**
- Classifications.
- Applications.

Introduction

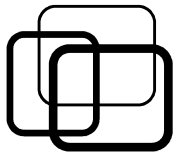


- Recursion concept:
  - What is recursion?

    ➔ Look up slides #6: “What is recursion?”!!
  - Recursion is…

    ➔ Defining a problem by itself!!
  - For example:
    - Factorial: n! = n * (n – 1)!.
    - Fibonacci: f(n) = f(n – 1) + f(n – 2).
    - Natural number: n is natural ⇔ n – 1 is also natural.
    - Ancestor: A’s ancestors are also A’s parents’ ancestors.

Introduction



- Recursion concept:
  - Meaningful recursion:
    - Base case: explicit definition.
    - Recursive case: reduce problem to simpler cases.

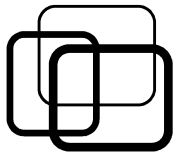
      - 0! = 1
      - n! = n * (n – 1)!.

      - f(0) = 0
      - f(1) = 1
      - f(n) = f(n – 1) + f(n – 2).

      - 0 is the smallest natural number
      - n is natural ⇔ n – 1 is natural.

      - Nearest A's ancestors are A's parents.
      - A's ancestors are also A's parents' ancestors.

Introduction

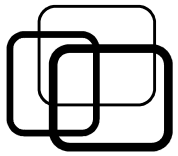


- Recursion in programming:
  - Recursive function:
    - Has a call to itself in function body.

```
// Direct call.
void func()
{
        // …
        func();
        // …
}
```

```
// Indirect call.
void func1()
{
        // …
        func2();
        // …
}
void func2()
{
        // …
        func1();
        // …
}
```

Introduction



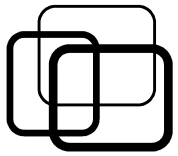
- Recursion in programming:
  - Meaningful recursive function:

```
<Return type> <Function name>( [Arguments] )
{
    if (<base case>)
        // solve explicit.
    else
        // call to itself with simpler arguments.
}
```

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

```
int fibo( int n )
{
    if (n == 0)
        return 0;
    if (n == 1)
        return 1;
    return fibo(n – 1) + fibo(n – 2);
}
```

Introduction



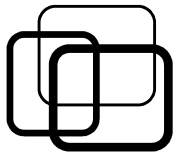
- Recursion in programming:
  - Recursive type:

```
struct <Struct name>
{
    // …
    <Struct name>  <member>;
    // …
};
```

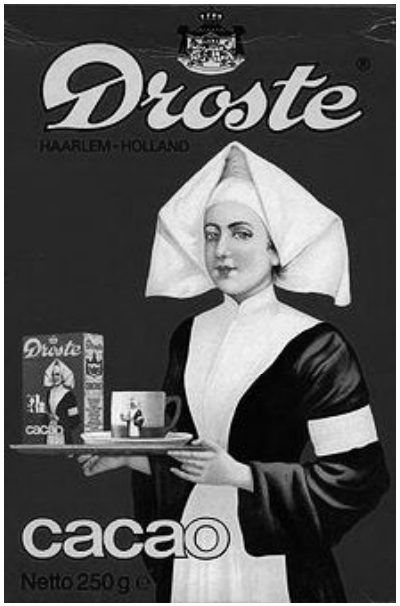
```
struct Person
{
    char    *name;
    int     age;
    Person *father;
    Person *mother;
};
```

```
struct Employee
{
    char        *name;
    char        *address;
    double      salary;
    Employee *manager;
};
```

Introduction

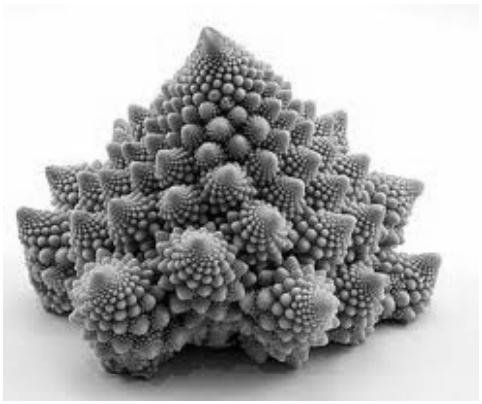


- Recursion in real life:



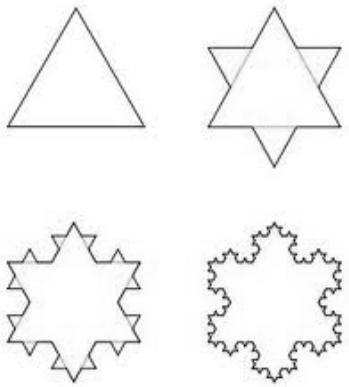
Marketing

Russian dolls



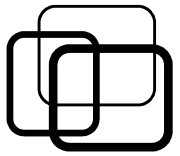
Nature

Graphics



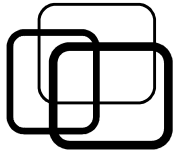
Leaf

Contents



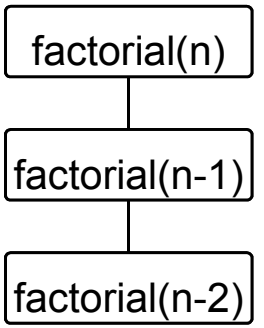
- Introduction.
- **Classifications.**
- Applications.

Classifications



- Types of recursion:
  - Linear recursion:
    - Function body has ONE call to itself.

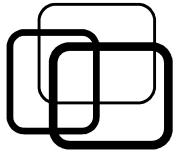
```
int factorial( int n )
{
    if (n == 0)
        return 1;
    return n * factorial( n – 1 );
}
```



    - Complexity: O(n).
  - Tail recursion:
    - Special case of linear recursion.
    - Recursive call is last statement.

➔ Transform to loop (do not need to store previous cases).

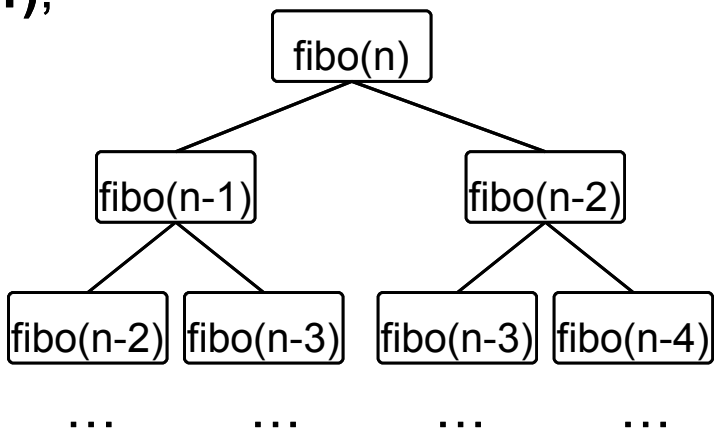
Classifications



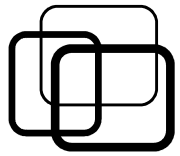
- Types of recursion:
  - Binary recursion:
    - Function body has TWO calls to itself.

```
int fibo(int n)
{
    if (n == 0)
        return 0;
    if (n == 1)
        return 1;
    return fibo(n – 1) + fibo(n – 1);
}
```

    - Complexity: $O(2^n)$.



Classifications



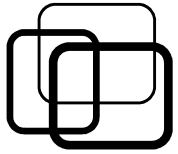
- Types of recursion:
  - Mutual recursion:
    - Function f1 has a call to f2.
    - Function f2 has a call to f1.

```
bool checkEven(int n)
{
    if (n == 0)
        return true;
    return checkOdd(n – 1);
}
```

```
bool checkOdd(int n)
{
    if (n == 0)
        return false;
    return checkEven(n – 1);
}
```

    - Complexity: depends on the functions.

Classifications

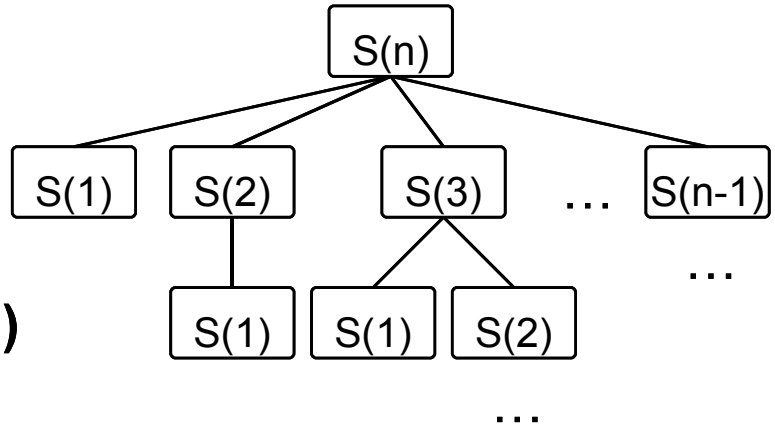


- Types of recursion:
  - Non-linear recursion:
    - Function body has call to itself INSIDE A LOOP.

Calculate: S(1) = 1

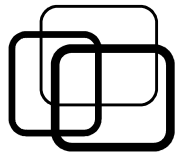
S(n) = S(1) + S(2) + … + S(n – 1).

```
int calculate( int n )
{
    if (n == 1)
        return 1;
    int S = 0;
    for (int i = 1; i <= n - 1; i++)
        S += calculate( i );
    return S;
}
```



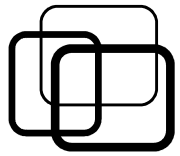
    - Complexity: O(n!).

Contents



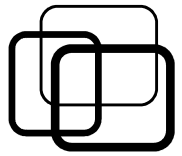
- Introduction.
- Classifications.
- **Applications.**

Applications



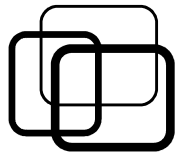
- Problem solving:
  - Non-recursive solution:
    - Direct solution.
    - Find steps to the problem.
  - Recursive solution:
    - Indirect solution.
    - Move the problem to us!
    - Re-define the problem recursively:
      - Step 1: solve base case directly.
      - Step 2: reduce the problem to simpler cases.
        - ➔ Regression formula (reduce by formula).
        - ➔ Divide and conquer (reduce by splitting).
        - ➔ Back-tracking (find all solutions).

Applications



- Regression formula:
  - Calculate element A_n in series { A }:
  - Regression formula:
    - Step 1: find direct formula to calculate A_0.
    - Step 2: find formula to calculate A_n from A_{n-1}.

Applications



- Regression formula:
  - Example 1:
    - Bacteria double in every min.
    - 1 bacteria at first.
    - How many after 20 mins?

      V(0) = 1
      V(n) = 2 * V(n – 1).

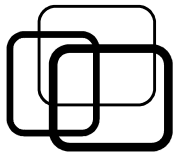
```
int calcBac( int n )
{
    if ( n == 0)
        return 1;
    return 2 * calcBac( n – 1 );
}
```

  - Example 2:
    - Saving rate: 7% / year.
    - Deposit 1 million.
    - How much after 20 years?

      T(0) = 1
      T(n) = T(n – 1) + 0.07 * T(n – 1)}
      = 1.07 T(n – 1).

```
int calcMoney( int k )
{
    if ( k == 0)
        return 1;
    return 1.07 * calcMoney(k -1);
```

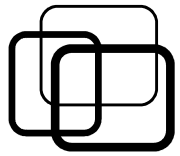
Applications



- Divide-and-conquer:
  - How to eat a cow?
    - ➔ Split into small parts.
    - ➔ How small is enough?
  - Divide-and-conquer technique:

```
Conquer ( P )
{
        if ( P is small enough )
                Solve P directly;
        else
                Split P → P1, P2;
                Conquer ( P1 );
                Conquer ( P2 );
                Join results;
}
```

Applications



- Divide-and-conquer:
  - Example:
    - Given array of integers.
    - Count negative numbers.

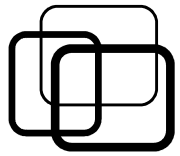
- Small array (1 element):
  + Check 1 element to count.
- Large array:
  + Split into 2 sub-arrays.
  + Count each sub-array recursively.
  + Sum the result.

```
int countNegs( int *a, int l, int r )
{
  if (l == r)
        return a[ r ] < 0 ? 1 : 0;

  int mid = ( l + r ) / 2;
  int c1 = countNegs(a, l, mid);
  int c2 = countNegs(a, mid+1, r);

  return c1 + c2;
}
```

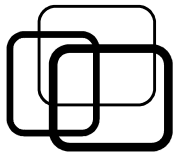
Applications



- Back-tracking:
  - Also called “trial-and-error”.
  - Find all solutions.

```
Try ( S )
{
    if (S is solution)
            Log S;
    else
            while has next step
            {
                Update S to next step;
                Try ( S );
                Roll back S;
            }
}
```

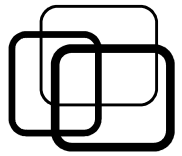
Applications



- Back-tracking:
  - Example:
    - Given array of positive integers.
    - Find all set of elements whose sum equal K.

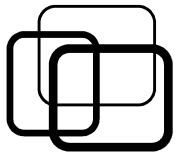
```
void findSet( int *a, int n, int K, int start, int T, bool *flag ) {
        if ( T == K )
                print( a, n, flag );
        else
                for ( int i = start; i < n; i++ )
                {
                        T += a[ i ];  flag[ i ] = true;
                        findSet( a, n, K, i + 1, T, flag );
                        T -= a[ i ];   flag[ i ] = false;
                }
}
```

Summary



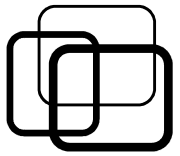
- Introduction:
  - Recursion is defining problem by itself.
  - Meaningful recursion:
    - Base case: explicit definition.
    - Recursive case: reduce problem to simpler cases.
  - Recursive function: has a call to itself.
  - Recursive type: has itself as member.
- Classifications:
  - Types: linear, binary, mutual, non-linear.
- Applications:
  - Regression formula, divide-conquer.

Summary



- Classifications:
  - Linear recursion: 1 recursive call.
  - Binary recursion: 1 recursive call.
  - Mutual recursion: 2 functions call each other.
  - Non-linear recursion: recursive call in loop.
- Applications:
  - Regression formula.
  - Divide-and-conquer.
  - Back-tracking.

Practice



- **Practice 6.1:**

  Write C/C++ recursive functions to do the followings:

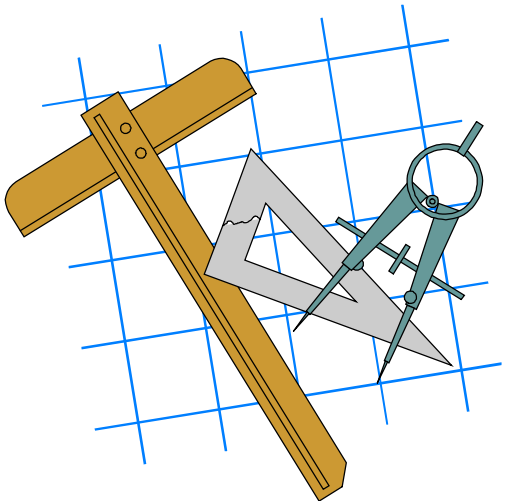
  a) Calculate A = 1 + 2 + … + n.

  b) Calculate B = x^n.

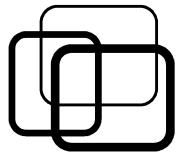
  c) Calculate C = 1/1 – 1/2 + 1/3 – 1/4 + … (+/-) 1/n.

  d) Print to screen Pascal Triangle with height = N.

```
1
1   1
1   2   1
1   3   3   1
1   4   6   4   1
…
```



Practice



- Practice 6.2:

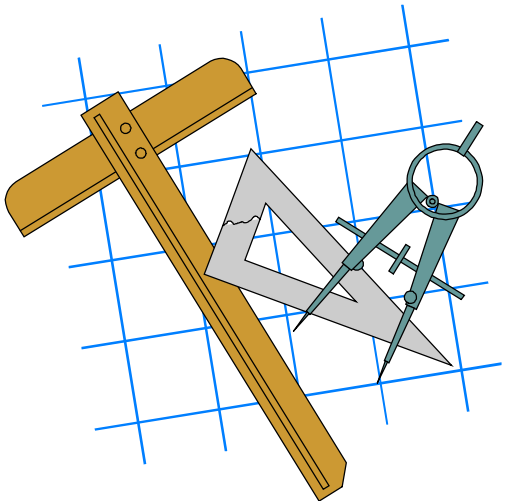
  Write C/C++ recursive functions to do the followings:

  a) Sum all even numbers in array.

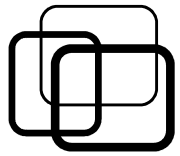
  b) Find max number in array.

  c) Reverse a string.

  d) Print currency format of a positive integer.



Practice



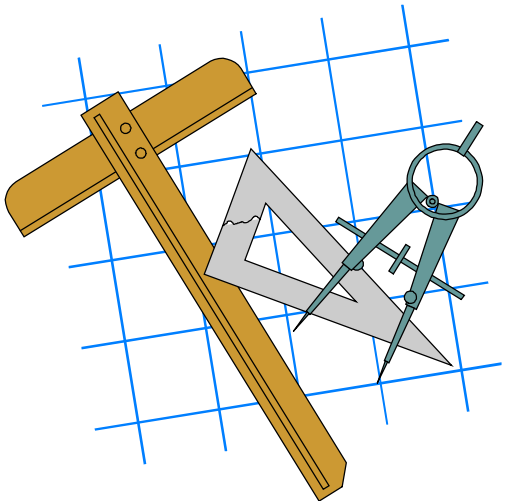
- Practice 6.3:

  Write the following recursive functions:

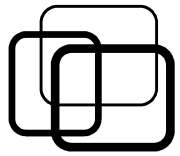
  a) Print all permutations from 1 to N.

  b) Print all K-permutations from 1 to N.

  c) Print all K-sets from 1 to N.



Practice



- Practice 6.4:

  Write C/C++ recursive functions to do the followings on struct **Person**:

  a) Count all ancestors of a given person.

  b) Count all ancestors from the mother-side of a given person.