RECURSION

PROGRAMMING TECHNIQUES

ADVISOR: Trương Toàn Thịnh

CONTENTS

INTRODUCTION

- Have a big significance in computer science
- Suitable for problem with recursive nature
- Limitations
  - Low speed
  - Need a large amount of memory
- For example: recursively defining a natural number
  - Zero (0) is a natural number
  - n is a natural number if $n - 1$ is a natural number

INTRODUCTION

- Example: recursively defining the factorial
  - 0! = 1
  - $n! = n * (n - 1)!$
- Program
  - long GT(int n){
    - long ret;
    - if(n == 0) ret = 1;
    - else ret = n * **GT(n – 1)**;
    - return ret;
  - }

INTRODUCTION

- Example: compute recursively
  - If $x = 0$ → result = 0
  - If $x < 0$ → result = -
  - If $x > 0$ → result =
- Program

```
double SQRT3(double x){
    double ret;
    if(x == 0) ret = 0;
    else {
        if(x < 0) ret = SQRT3(-x);
        else ret = pow(x, 1.0/3);
    }
    return ret;
}
```

INTRODUCTION

- Example: recursively report an error
  - If *len*(*string-to-print*) > 50 → print *error-string*
  - Else print *string-to-print*
- Program
  - void printString(char* s){
    - if(strlen(s) <= 50) cout << s << endl;
    - else printError();
  - }
  - void printError(){
    - printString("String exceeding limited length");
  - }

INTRODUCTION

- Stop condition
  - At ex1: at base step $n = 0$ → result is 1
  - At ex2:
    - At base step $x = 0$ → result is 1
    - At recursive step $x > 0$ → compute normally
  - At ex3:
    - Error-string obeys the limited length
- What if stop condition is wrong
  - Loop forever
  - Stack overflown

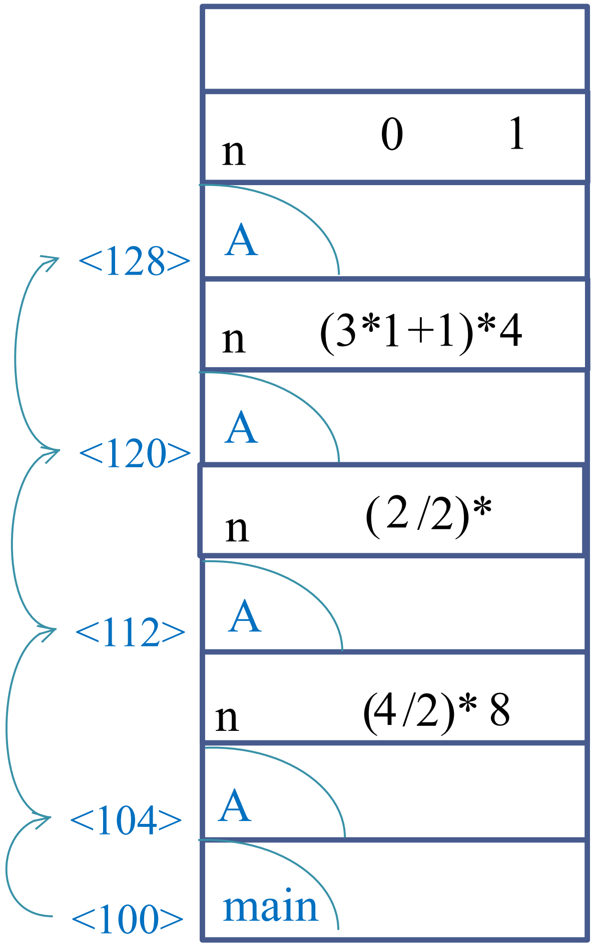
CATEGORIES

- Linear recursion
  - In the function's body, there is only one function-call to itselt directly
  - General form
    - <Function-name & parameters list>{
      - if(<stop-condition>)
        - **/*Return a value or stop working*/**
      - else{
        - **/*Do something*/**
        - **/*Call recursively*/**
      - }
    - }

CATEGORIES

- Linear recursion
  - Example: Consider $\{a_n\}$, n ⧧ 0 with following rule
    - If $n = 0$ then $a_0 = 1$
    - If n even then $a_n = (n/2)*a_{n/2}$
    - If n odd then $a_n = (3n + 1)*a_{n-1}$
  - Program

```
int A(int n){
    if(n <= 0) return 1;
    else if(n % 2 == 0)
        return (n/2)*A(n/2);
    else
        return (3*n + 1)*A(n – 1);
}
void main(){
    cout << A(4) << endl;
}
```



CATEGORIES

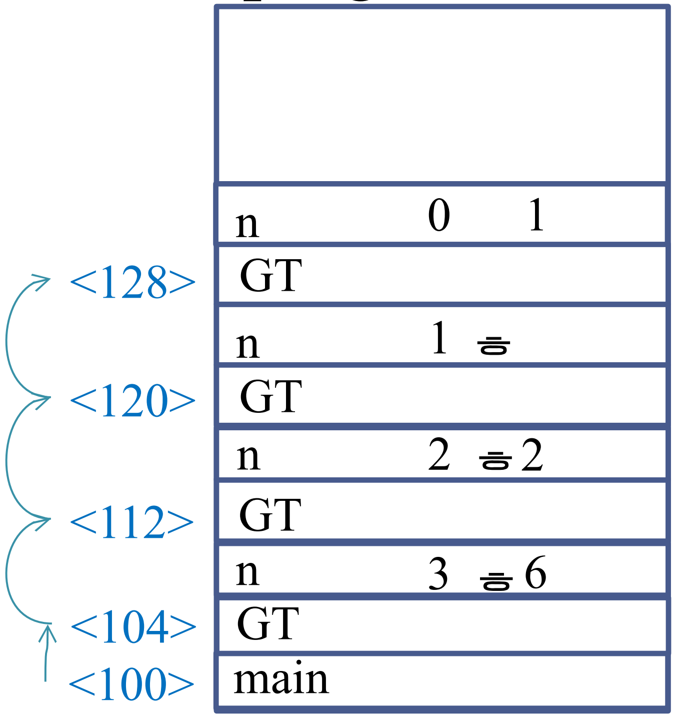
- Linear recursion
  - The last call (tail call): When function g() call function f(), we say f() is a tail-call if the f()'s termination is the termination of g()
  - Tail-call does not mean it is at the last-line of the function's body

float CalcAB(float t){	float Calc(float t){
float y = FuncA(t);	float y = FuncA(t);
float x = FuncB(t);	float x = FuncB(t);
return **Max**(t*x, t*y);	if(x > y) return **FuncA**(x – y);
}	return (y – x)*Max(t*x, t*y);
	}

CATEGORIES

- Linear recursion
  - Tail recursion is linear recursion, and have a recursive call which is a tail-call
  - Example: **NOT** tail-recursion program

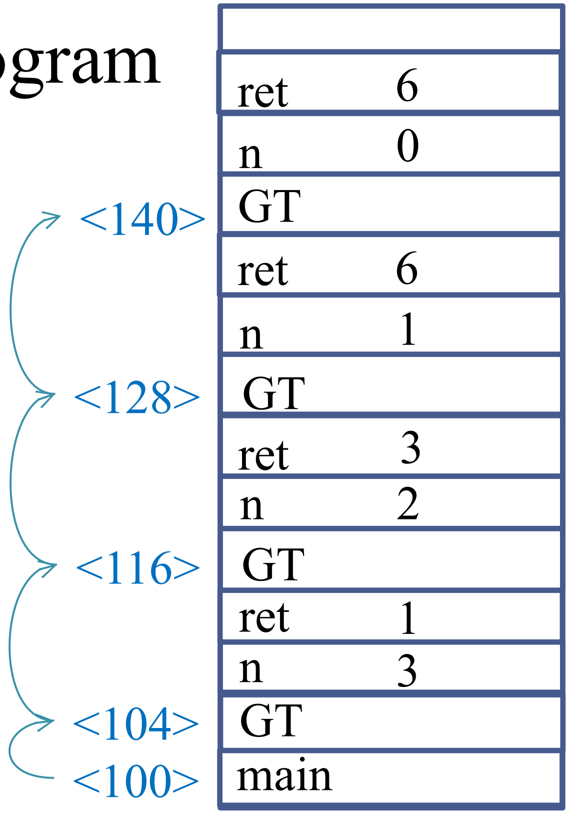
```
void main(){
    cout << GT(3) << endl;
}
long GT(int n){
    if(n == 0) return 1;
    return n * GT(n – 1);
}
```



CATEGORIES

- Linear recursion
  - Tail recursion is linear recursion, and have a recursive call which is a tail-call
  - Example: tail-recursion program

```
void main(){
    cout << GT(3) << endl;
}
long GT(int n, long ret = 1){
    if(n == 0) return ret;
    return GT(n – 1, ret * n);
}
```



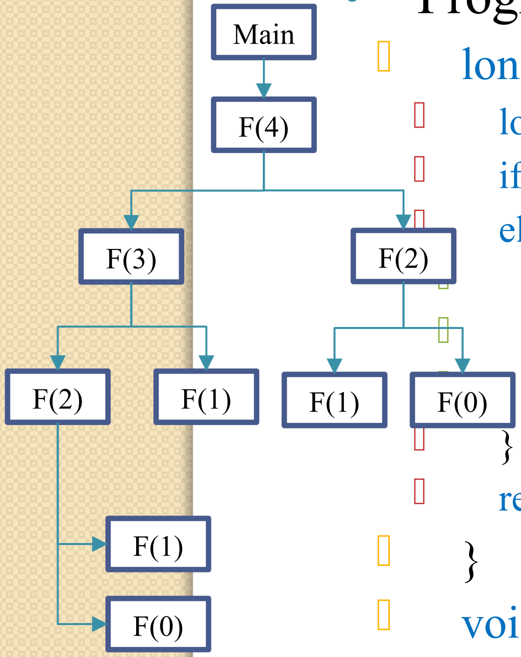
CATEGORIES

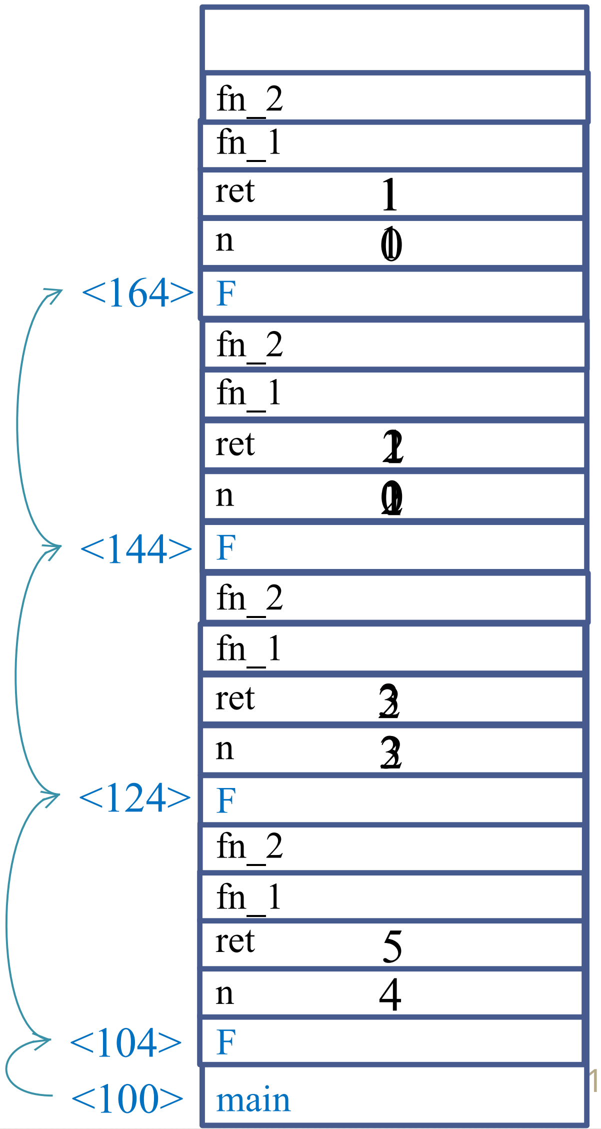
- Binary recursion
  - In the function's body, there are exact 2 recursive call directly
  - General form
    - <Function name and parameters list>{
      - if(<stop condition>)
        - **/*Return value or stop working*/**
      - else{
        - **/*Do something*/**
        - **/*Recursively call (1) to solve the smaller problems*/**
        - **/*Recursively call (2) to solve the remaining problems*/**
      - }
    - }

CATEGORIES

- Binary recursion
  - Ex: Consider Fibonaci $\{F_n\}$, $n \geq 2$
    - If $n = 0 \vee n = 1$ then $F_0 = F_1 = 1$
    - If $n \geq 2$ then $F_n = F_{n-1} + F_{n-2}$
  - Program

```
long F(int n){
    long ret, fn_1, fn_2;
    if(n <= 1) ret = 1;
    else{
        fn_1 = F(n – 1);
        fn_2 = F(n – 2);
        ret = fn_1 + fn_2;
    }
    return ret;
}
void main(){
    cout << F(4) << endl;
}
```

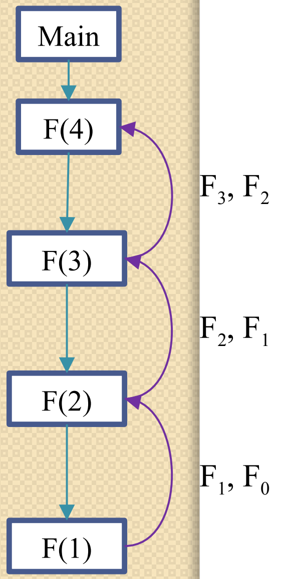


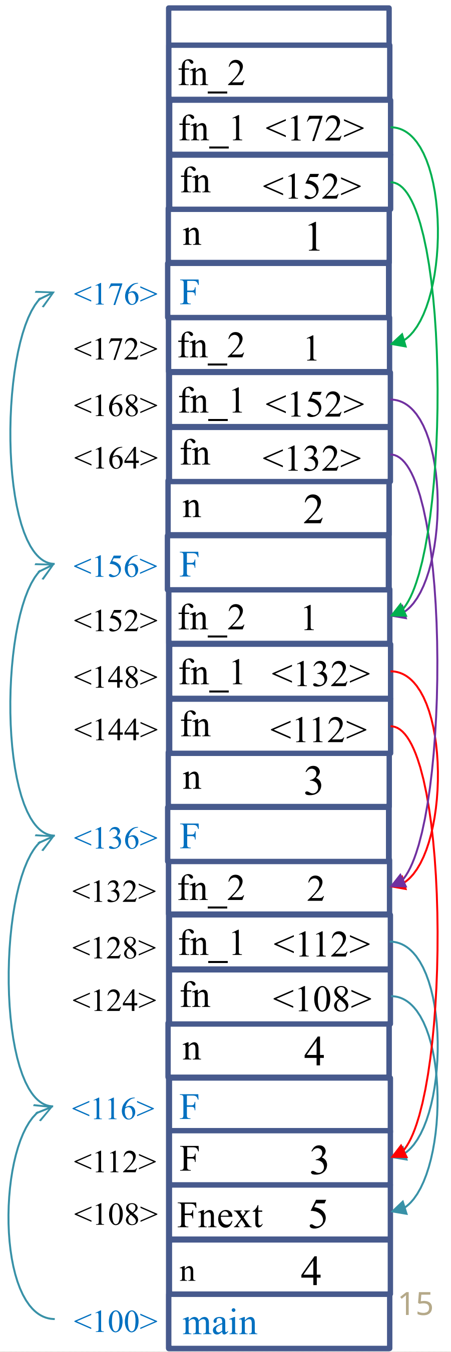


CATEGORIES

- Binary recursion
  - Ex: Consider Fibonaci $\{F_n\}$, $n \geq 2$
    - If $n = 0 \vee n = 1$ then $F_0 = F_1 = 1$
    - If $n \geq 2$ then $F_n = F_{n-1} + F_{n-2}$
  - Program improved

```
void F(int n, long* fn_1, long* fn){
    long fn_2;
    if(n <= 1) *fn_1 = *fn = 1;
    else{
        F(n – 1, &fn_2, &(*fn_1));
        *fn = *fn_1 + fn_2;
    }
}
void main(){
    long F, Fnext;
    cout << F(4, &F, &Fnext) << endl;
}
```

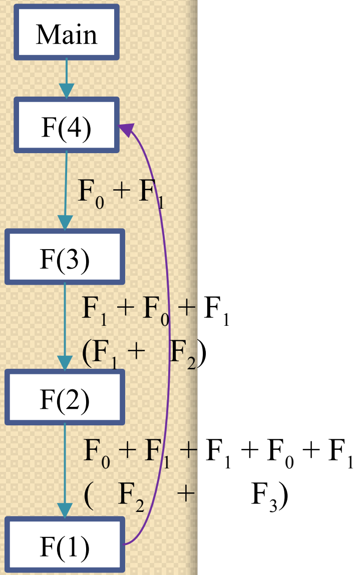


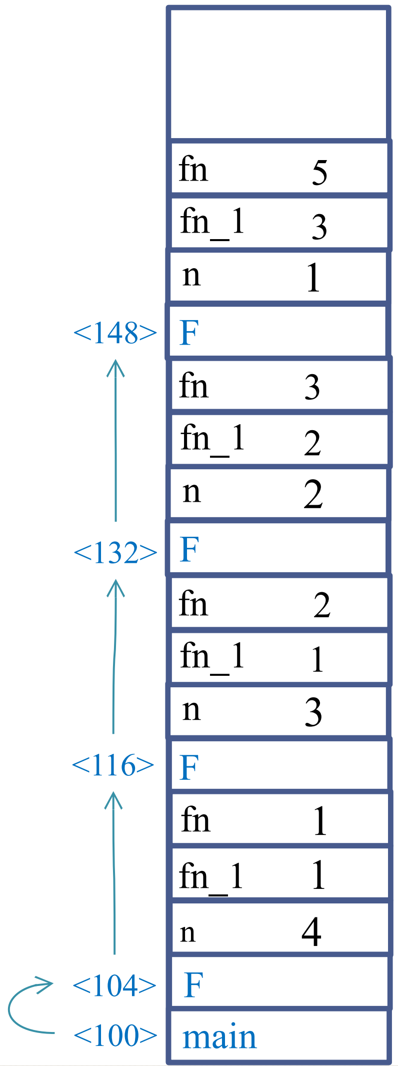


CATEGORIES

- Binary recursion
  - Ex: Consider Fibonaci $\{F_n\}$, $n \geq 2$
    - If $n = 0 \vee n = 1$ then $F_0 = F_1 = 1$
    - If $n \geq 2$ then $F_n = F_{n-1} + F_{n-2}$
  - Program tail-recursion

```
long F(int n, long fn_1 = 1, long fn = 1){
    if(n <= 1)
        return fn;
    else
        return F(n – 1, fn, fn_1 + fn);
}
void main(){
    cout << F(4) << endl;
}
```





CATEGORIES

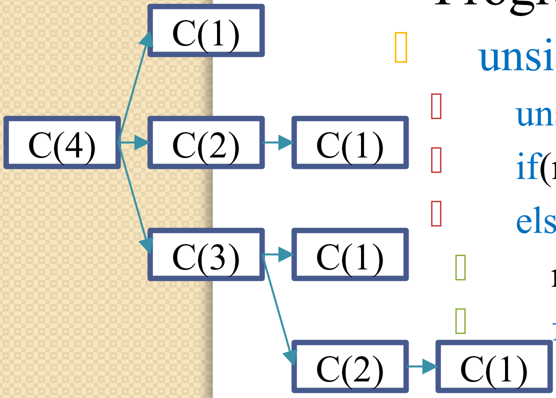
- Non-linear recursion
  - Recursively call is in a loop. It can be said that non-linear recursion is a general form of binary recursion
  - General form

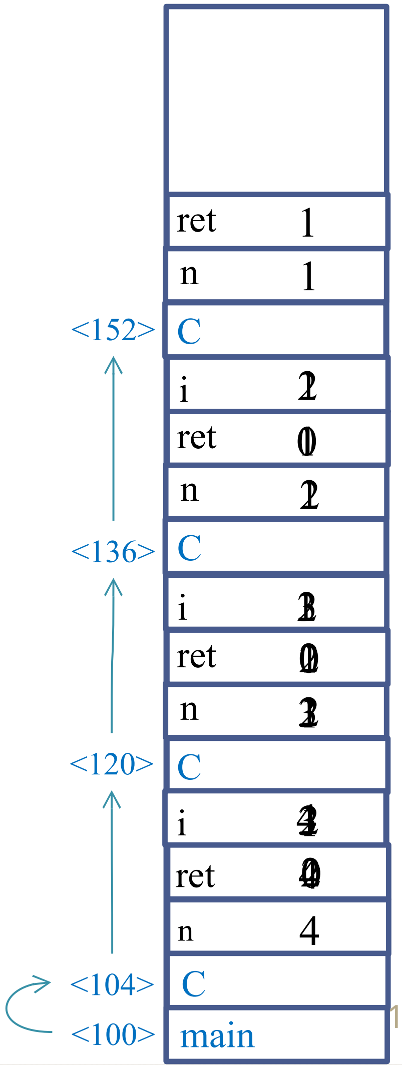
```
<Function name and parameters list>{
    if(<stop-condition>)
        /*Return a value or stop working*/
    else{
        loop {
            /*Do something*/
            /*Recursively call to solve the smaller problems*/
        }
    }
}
```

CATEGORIES

- Non-linear recursion (nesting recursion)
  - Example:
    - $C_1 = 1$ when $n = 1$
    - $C_n = C_1 + C_2 + \ldots + C_{n-1}$ when $n > 1$
  - Program

```
unsigned long C(int n){
    unsigned long ret;
    if(n == 1) ret = 1;
    else{
        ret = 0;
        for(int i = 1; i < n; i++)
            ret += C(i);
    }
    return ret;
}
void main(){
    cout << C(4) << endl;
}
```





CATEGORIES

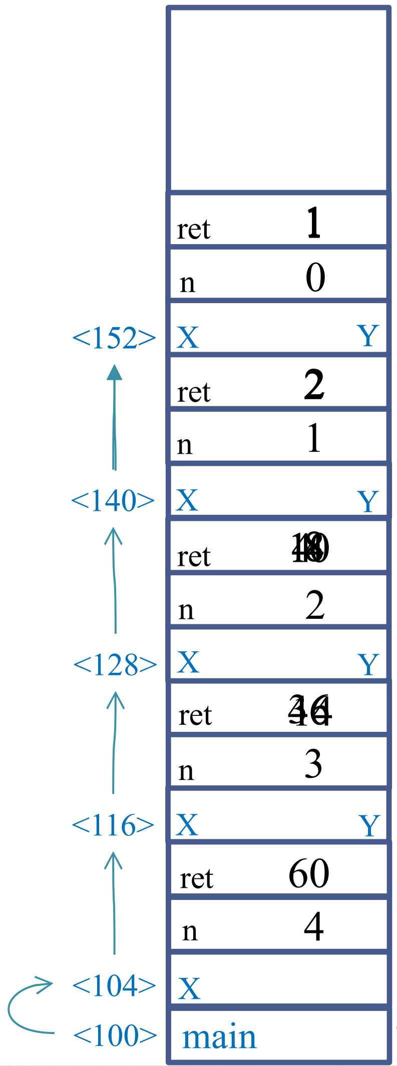
- Mutual recursion
  - Indirectly recursive call is through another different function
  - May change to different form of recursion
  - General form

```
<1st function name and parameters list>{
    if(<stop-condition>)
        /*Return a value or stop working*/
    else{
            /*Do something*/
            /*Call to 2nd function*/
        }
    }
}
<2nd function name and parameters list>{
    if(<stop-condition>)
        /*Return a value or stop working*/
    else{
            /*Do something*/
            /*Call to 1st function*/
        }
    }
}
```

CATEGORIES

- Mutual recursion
  - Example:
    - $X_0 = Y_0 = 0;$
    - $X_n = X_{n-1} + Y_{n-1};$
    - $Y_n = n^2$ ㅎ $X_{n-1} + Y_{n-1}$
  - Program

```
long X(int n){
    long ret;
    if(n <= 0) ret = 1;
    else ret = X(n – 1) + Y(n – 1);
    return ret;
}
long Y(int n){
    long ret;
    if(n <= 0) ret = 1;
    else ret = n*n*X(n – 1) + Y(n – 1);
    return ret;
}
void main(){ cout << X(4) << endl; }
```

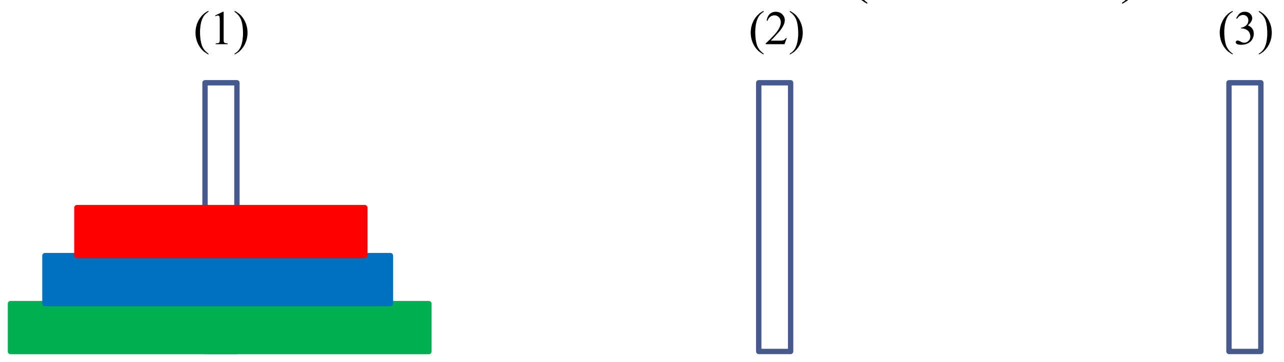


SOME APPLICATIONS

- Can use recursion to solve some problems
- In some cases, recursion is a simple and easy-to-understand method
- This method consumes time and memory very much
- Carefully pay attention to stop-condition
- Some problems: Ha-Noi tower, recurrence formula, combination, permutation, find the biggest/smallest, sort

SOME APPLICATIONS

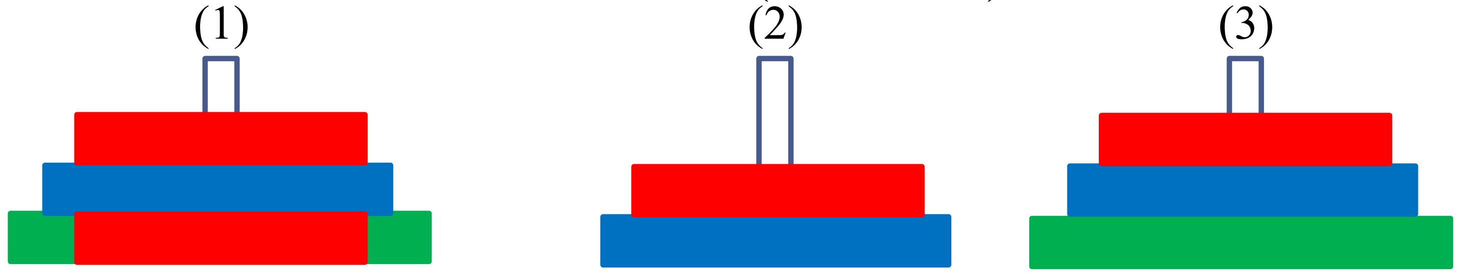
- Ha Noi tower with 3 column ($n = 3$ disc)



  - Description:
    - Have three columns (1, 2, 3)
    - At first, 1st column has 3 disc
    - Move disc from 1st column to 3rd column (2nd column is intermerdiate), such that **bigger disc is under smaller one** and **move only one disc at one time**
  - Recursive method
    - Move upper $n - 1$ disc from 1st column to 2nd column
    - Move the last disc from 1st column to 3rd column
    - Move $n - 1$ disc from 2nd column to 3rd column

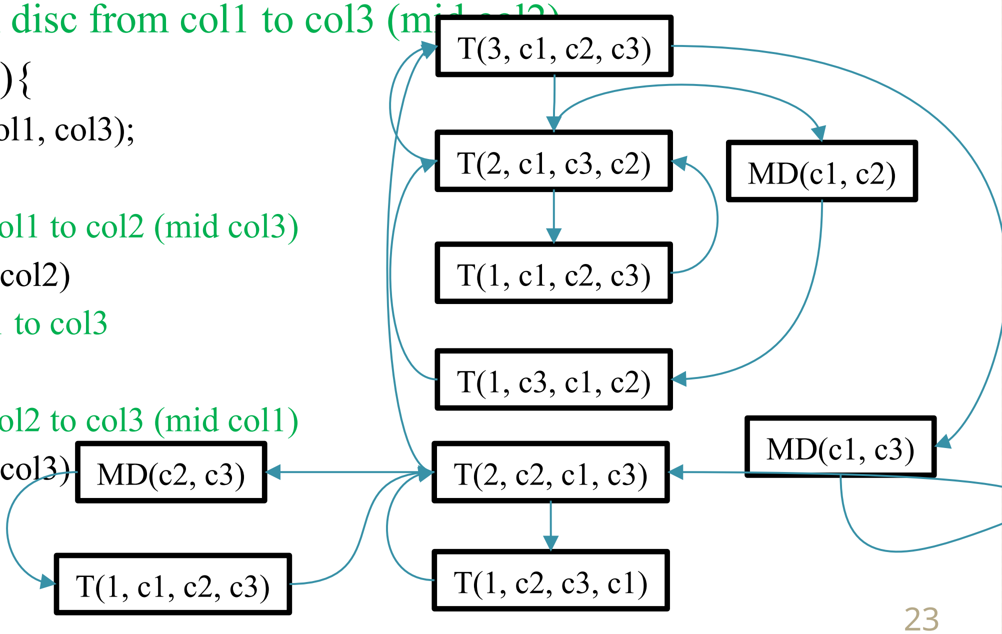
SOME APPLICATIONS

- Ha Noi tower with 3 column (n = 3 disc)



  - Pseudo-code

```
// Using Tower to move n disc from col1 to col3 (mid col2)
Tower(n, col1, col2, col3){
    If(n = 1) Then MoveDisc(col1, col3);
    Else{
        //Move n – 1 disc from col1 to col2 (mid col3)
        Tower(n – 1, col1, col3, col2)
        //Move disc nth from col1 to col3
        MoveDisc(col1, col3);
        //Move n – 1 disc from col2 to col3 (mid col1)
        Tower(n – 1, col2, col1, col3)
    }
}
```



SOME APPLICATIONS

- Recurrence formula
  - Usually in mathematics and approximate computation
  - May directly change induction formula to recursive program
  - Example 1:
    - Virus doubles the amount after one hour. How many viruses are there after *h* hours
      - $V_0 = 2$ (At first there are 2 viruses)
      - $V_h = 2 \times V_{h-1}$
    - Program

```
long ComputeVirus(int h){
    if(h == 0) return 2;
    return 2 * ComputeVirus(h – 1);
}
```

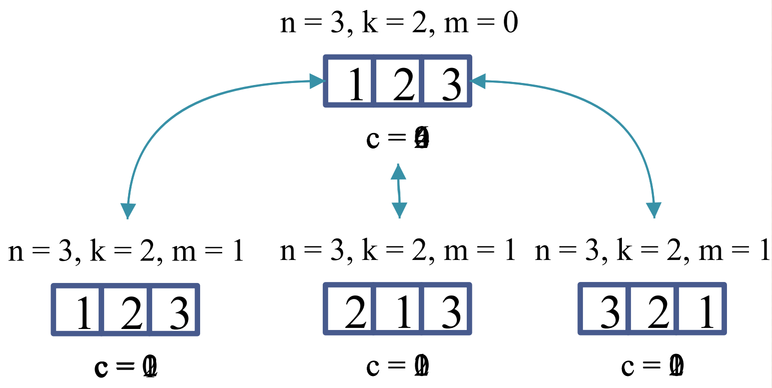
  - Example 2:
    - Bank rate is 14%/year. At first, a sender sends 1000000, How many sum of the money is there after *n* years
      - $T_0 = 1000000$
      - $T_n = T_{n-1} + 14\% \times T_{n-1} = 1.14 \times T_{n-1}$
    - Program

```
double ComputeMoney(int n){
    if(n == 0) return 1000000;
    return 1.14*ComputeMoney(n – 1);
}
```

SOME APPLICATIONS

- Combination / Permutation
  - Example: we have = {1, 2, 3}
    - There are six permutations: 123, 132, 213, 231, 312, 321
    - There are six 2-permutation of 3: 12, 13, 21, 31, 23, 32
  - Formula: ${}_nP_n = n!$ và ${}_nP_k = n!/(n - k)!$
- Program

```
long P(char *a, int n, int k, int m = 0){
    long c = 0;
    for (int i = m; i <= (n - 1); i++){
        swap(a[m], a[i]);
        if(m < k - 1) c += P(a, n, k, m + 1);
        else{
            for(int t = 0; t < k; t++) cout << a[t];
            cout << endl;
            c++;
        }
        swap(a[m], a[i]);
    }
    return c;
}
void main(){cout << P("123", 3, 2);}
```



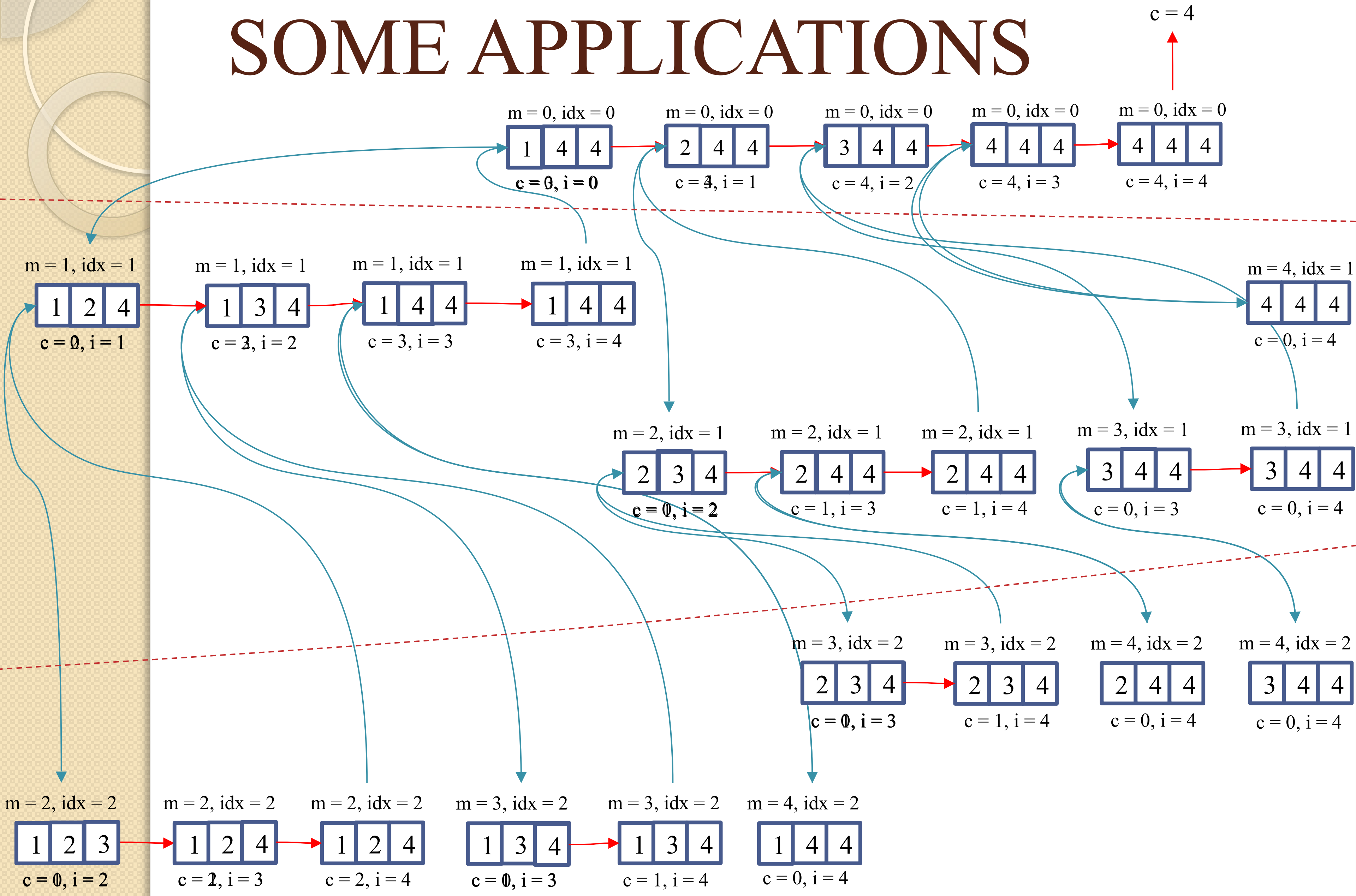
12 13 21 23 32 31

SOME APPLICATIONS

- Combination
  - Consider ▭ = {1, 2, 3}, there're three 2-combination of 3: 12, 13, 23
  - Formula: ${}_nC_n = 1$ và ${}_nC_k = n!/(k!$ ㅎ $(n - k)!)$
  - Program (demo slide 27)

```
long C(char *a, char* kq, int n, int k, int m = 0, int idx = 0){
    long c = 0;
    for (int i = m; i < n; i++){
        kq[idx] = a[i];
        if(idx < k - 1)
            c += C(a, kq, n, k, i + 1, idx + 1);
        else{
            cout << kq << endl;
            c++;
        }
    }
    return c;
}
void main(){ cout << C("1234", "xxx", 4, 3); }
```

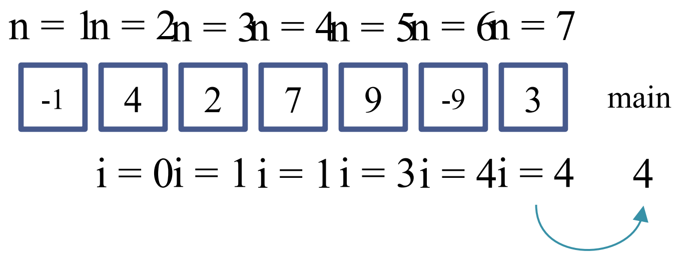
SOME APPLICATIONS



SOME APPLICATIONS

- Find the biggest/smallest element
  - Ex: a = {-1, 4, 2, 7, 9, -9, 3}. Find the position of the max number.
  - Idea:
    - If the array is empty then return -1
    - If the array have one element, then return
    - Else
      - Find the position of the biggest element among $n - 2$ members
      - Compare that position with $n - 1$ to determine which is the most suitable one
  - Program

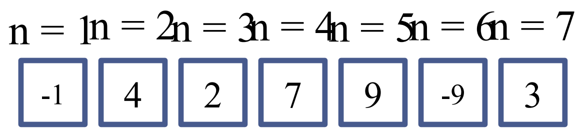
```
int csmax(int a[], int n){
    if(n <= 0) return -1;
    else if (n == 1) return 0;
    else{
        int i = csmax(a, n - 1);
        if(a[i] < a[n - 1]) return n – 1;
        return i;
    }
}
void main(){
    int a[] = {-1, 4, 2, 7, 9, -9, 3};
    cout << csmax(a, 7) << endl;
}
```



SOME APPLICATIONS

- Recursively sort (brute-force)
  - Example: Sort a = {-1, 4, 2, 7, 9, -9, 3} with increasing order.
  - Idea
    - If length of the array is greater than one
      - Sort *n* – 1 members with increasing order
      - Compare the last member with penultimate one of the sub-array just sorted
      - If the last member < penultimate one
        - Swap them
        - Re-sort the sub-array
  - Program

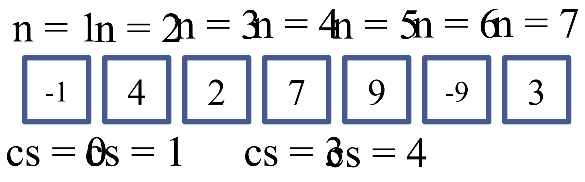
```
void sxTang(int a[], int n){
    if(n > 1){
        sxTang(a, n - 1);
        if(a[n - 1] < a[n - 2]){
            swap(a[n - 1], a[n - 2]);
            sxTang(a, n - 1);
        }
    }
}
void main(){
    int a[] = {-1, 4, 2, 7, 9, -9, 3};
    sxTang(a, 7);
}
```



SOME APPLICATIONS

- Recursively sort with csmax (brute-force)
  - Example: Sort a = {-1, 4, 2, 7, 9, -9, 3} with increasing order.
  - Idea
    - If length of the array is greater than one
      - Find the max position of *n* – 1 member
      - Compare the *a*[*n* – 1] with *a*[csmax] to swap if necessary
      - Re-sort the sub-array with *n* – 1 members
  - Program

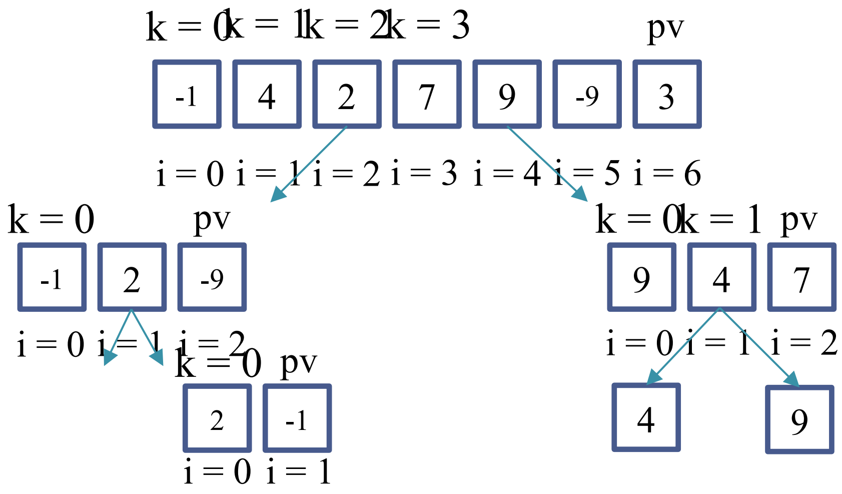
```
void sxTang(int a[], int n){
    if(n > 1){
        int cs = csmax(a, n - 1);
        if(a[n - 1] < a[cs]) swap(a[n - 1], a[cs]);
        sxTang(a, n - 1);
    }
}
void main(){
    int a[] = {-1, 4, 2, 7, 9, -9, 3};
    sxTang(a, 7);
}
```



SOME APPLICATIONS

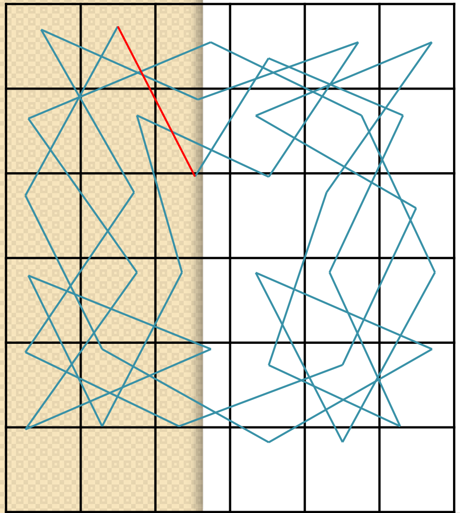
- Quick sort
  - Example: Sort a = {-1, 4, 2, 7, 9, -9, 3} with increasing order.
  - Idea
    - Choose the pivot (any position)
    - Process such that the left elements of pivot are smaller than pivot
    - Process such that the right elements of pivot are greater than pivot
    - Recursively process for 2 sub-array (exclude pivot)
  - Program

```
void QSort(int a[], int le, int ri){
    if(l >= r) return;
    int k = le;
    for(int i = le; i < ri; i++){
        if(a[i] <= a[ri]){
            swap(a[i], a[k]); k++;
        }
    }
    swap(a[k], a[ri]);
    QSort(a, le, k - 1);
    QSort(a, k + 1, ri);
}
void main(){
    int a[] = {-1, 4, 2, 7, 9, -9, 3};
    QSort(a, 0, 6);
```

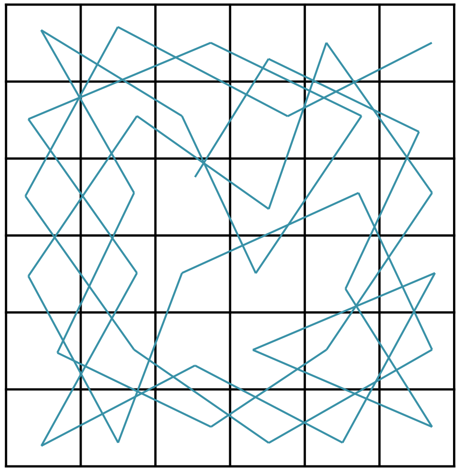


SOME APPLICATIONS

- Knight's tour (board size 6 ㅎ 6)
  - Description: Starting from cell (2, 2), let’s find the path (follow knight’s rule) to pass all the cells in board
  - There are 2 kinds of solutions: closed path and normal path
  - Board size and starting cell affect the amount of solutions (paths)



15	**36**	27	2	17	8
26	19	16	9	28	3
35	14	**1**	18	7	10
22	25	20	31	4	29
13	34	23	6	11	32
24	21	12	33	30	5

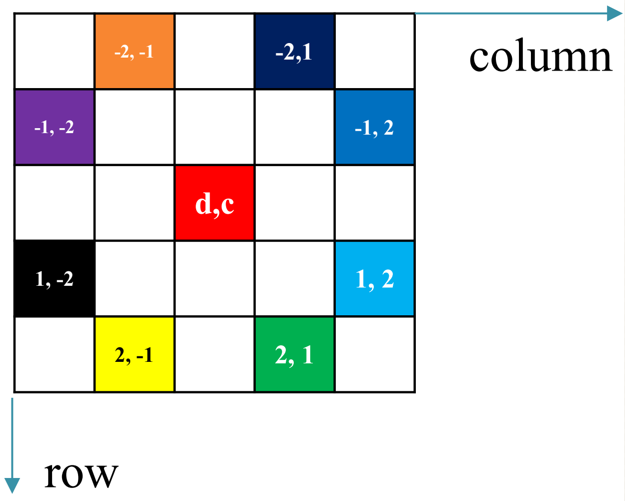


17	34	13	2	23	**36**
12	25	16	35	14	3
33	18	**1**	24	29	22
26	11	28	15	4	7
19	32	9	6	21	30
10	27	20	31	8	5

SOME APPLICATIONS

- Knight's tour
  - Hint:

```
#define SIZE 5
int dd[] = {-2, -1, 1, 2, 2, 1, -1, -2};
int dc[] = {1, 2, 2, 1, -1, -2, -2, -1};
int Bc[SIZE][SIZE]  ={0};
void NuocDi(int n, int d, int c){
    Bc[d][c] = n;
    if(n == MAX * MAX) xuatBc();
    for(int i = 0; i < 8; i++){
        int dmoi = d + dd[i], cmoi = c + dc[i];
        if(dmoi >= 0 && dmoi < MAX && cmoi >= 0 && cmoi < MAX && Bc[dmoi]
        [cmoi] == 0)
            NuocDi(n + 1, dmoi, cmoi);
    }
    Bc[d][c] = 0;
}
void main(){ NuocDi(1, 0, 0); }
```

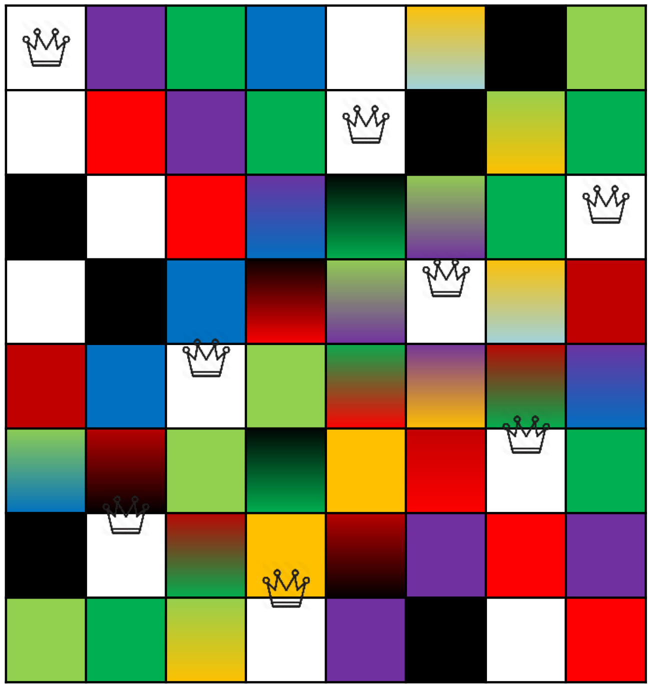


SOME APPLICATIONS

- Eight queens puzzle
  - Description: Need to put 8 queens into 8 ㅎ 8 board such that no queen can win the others (follow Chess's rule)
  - Note:
    - Name 8 queens as follows: 0, 1, 2, 3, 4, 5 ,6 ,7
    - Surely 8 queens is in 8 different rows ㅁ putting 0th queen in the 0th row
    - Need to find column indexes corresponding 0th row
    - Pay attention to main/sub diagonal after choosing row and column

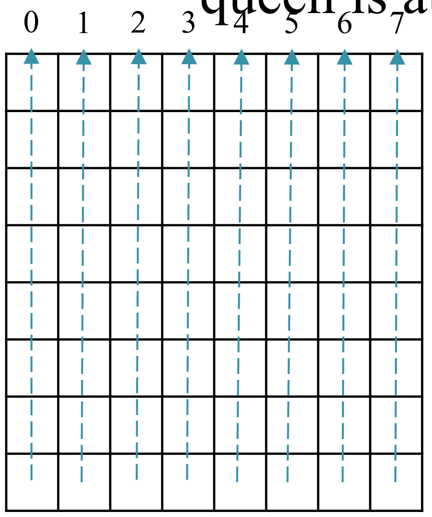
SOME APPLICATIONS

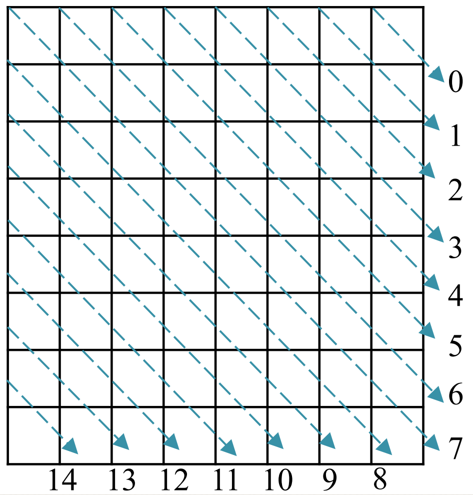
- Eight queens puzzle
  - Example of one solution

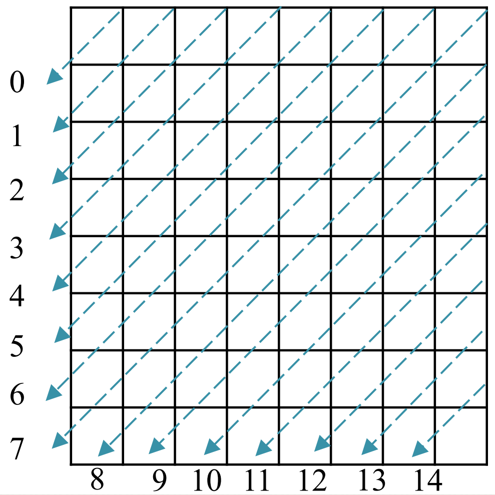


SOME APPLICATIONS

- Eight queens puzzle (program)
  - Hint:
    - Create array of columns, main/sub diagonal:
      - int ct[8] = {1, 1, 1, 1, 1, 1 ,1, 1};
      - int cxt[15] = { 1, 1, 1, 1, 1, 1 ,1, 1, 1, 1, 1, 1, 1, 1, 1 };
      - int cnt[15] = { 1, 1, 1, 1, 1, 1 ,1, 1, 1, 1, 1, 1, 1, 1, 1 };
    - Create array of solutions int lg[8]. It means that the *i*th queen is at *i*th row and lg[i]th column

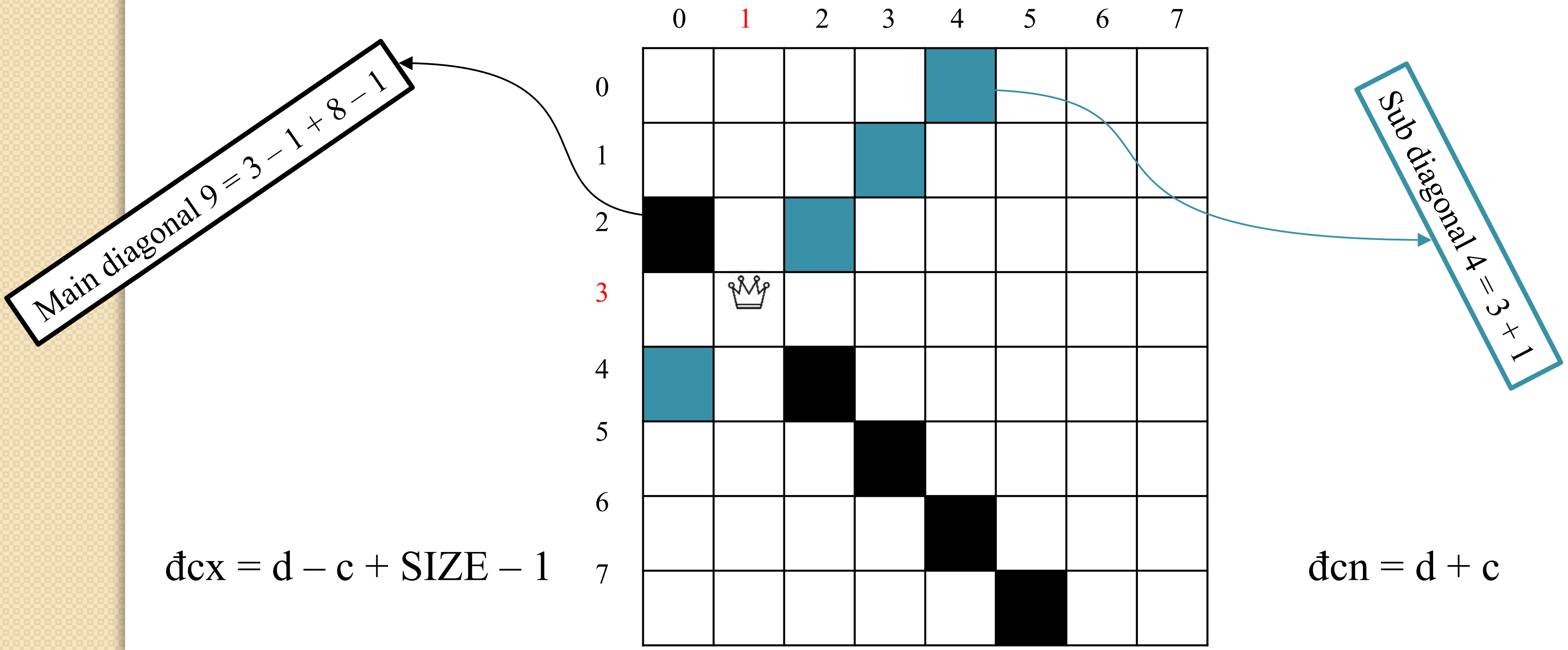






SOME APPLICATIONS

- Eight queens puzzle (program)
  - Hint:
    - Need determining the indexes of main/sub diagonals corresponding to row/col (i, j)'s (e.g., $i = 3$ & $j = 1$)



SOME APPLICATIONS

- Eight queens puzzle
  - Program:

```
void QH(int i){ // Put ith queen into ith row
    for(int j = 0; j < 8; j++){
        if(ct[j] && cxt[j - i + 8 - 1] && cnt[i + j]){
            lg[i] = j;
            ct[j] = cxt[j - i + 8 - 1] = cnt[i + j] = FALSE;
            if(i == 8 - 1) //print array lg
            else QH(i + 1);
            ct[j] = TRUE;
            cxt[j - i + 8 - 1] = TRUE;
            cnt[i + j] = TRUE;
        }
    }
}
void main(){
    QH(0);
}
```

	0	1	2	3	4	5	6	7
0	♕	x	x	x	x		x	x
1	x	x	♕	x	x	x	x	x
2	x	x	x	x	♕	x	x	x
3	x	♕	x	x	x	x	♕ x	♕ x
4	x	x	x	♕	x	x	x	♕ x
5	x	x	x	x	x	x	x	x
6	x	x	x	x	x	x	x	x
7	x	x	x	x	x	x	x	x

SOME APPLICATIONS

- Compute
  - Sum of the all members in array

```
long sum(int a[], int n){
    if(n < 1) return 0;
    return sum(a, n - 1) + a[n – 1];
}
```

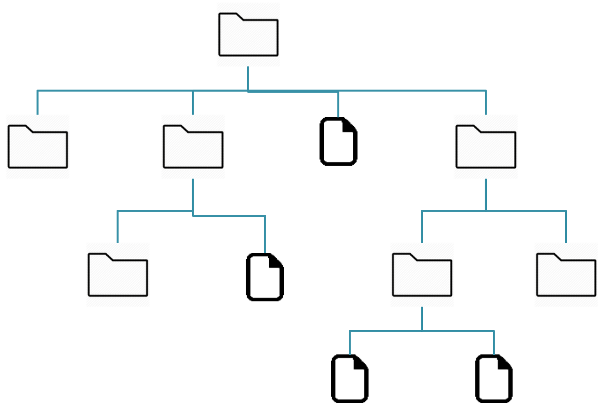
  - Power

Note: $x^7 = x^3$ ㅎ x^3 ㅎ x, $x^6 = x^3 * x^3$, $x^{-3} = 1/x^3$.

```
float LT(float x, int n){
    float ret, xlast;
    if(n < 0) ret = 1/LT(x, -n);
    else if (n == 0) ret = 1;
    else if (n == 1) ret = x;
    else {
        if(n % 2 == 0) { xlast = LT(x, n/2); ret = xlast * xlast; }
        else { xlast = LT(x, (n – 1)/2); ret = xlast * xlast * x;}
    }
    return ret;
}
```

SOME APPLICATIONS

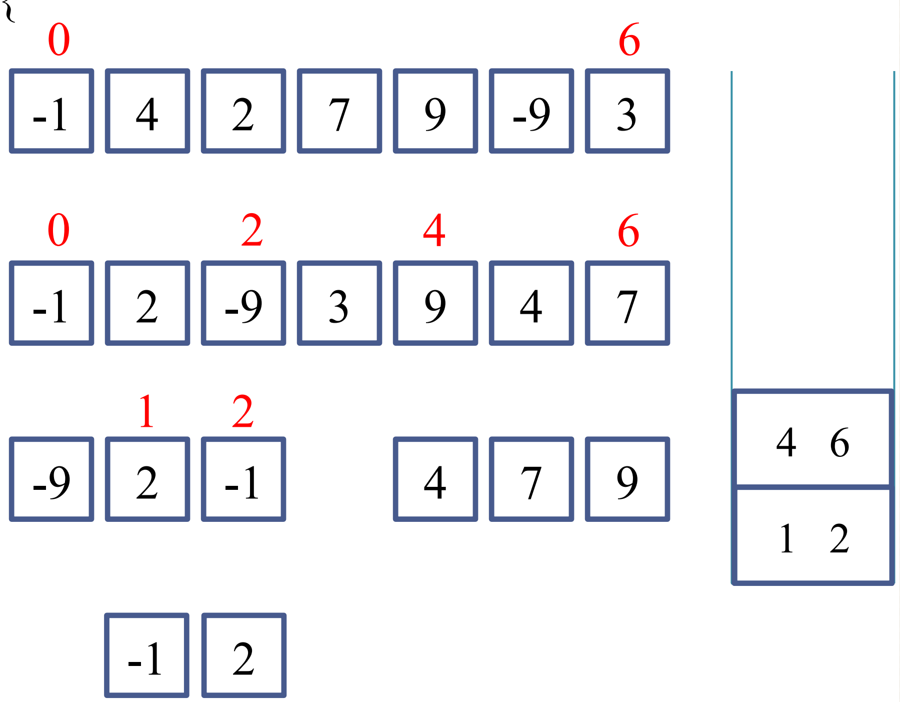
- Value of combination
  - = 1 when k = 0 hoặc k = n
  - = + when 0 < k < n
  - long C(int k, int n){
    - if(k == 0 || k == n) return 1;
    - return C(k, n - 1) + C(k – 1, n – 1);
  - }
- Size of directory tree
  - typedef struct tagFileSystem{
    - char* szName; bool isFile;
    - long nSize; int nSub;
    - tagFileSystem** paSub;
  - } FileSystem;
  - long getSize(FileSystem* f){
    - long nTotal = 0;
    - if(fs->isFile) return f->nSize;
    - for(int i = 0; i < f->nSub; i++) nTotal += getSize(f->paSub[i]);
    - return nTotal;
  - }



ALTERNATIVE METHOD

- There are 2 way of replacing recursion: loop and (stack+loop)
- Example: QuickSort with (stack+ loop)

```
typedef struct {int L, R; } Pair;
void QSort(int a[], int le, int ri){
    stack<Pair> s;
    Pair p = {le, ri};
    s.push(p);
    while(!s.empty()) {
        p = s.top(); s.pop();
        int k = p.L;
        for(int i = p.L; i < p.R; i++){
            if(a[i] <= a[p.R]){
                swap(a[i], a[k]); k++;
            }
        }
        swap(a[k], a[p.R]);
        if(p.L < k – 1) s.push({p.L, k - 1});
        if(p.R > k + 1) s.push({k + 1, p.R});
    }
}
void main(){ int a[] = {-1, 4, 2, 7, 9, -9, 3}; QSort(a, 0, 6); }
```

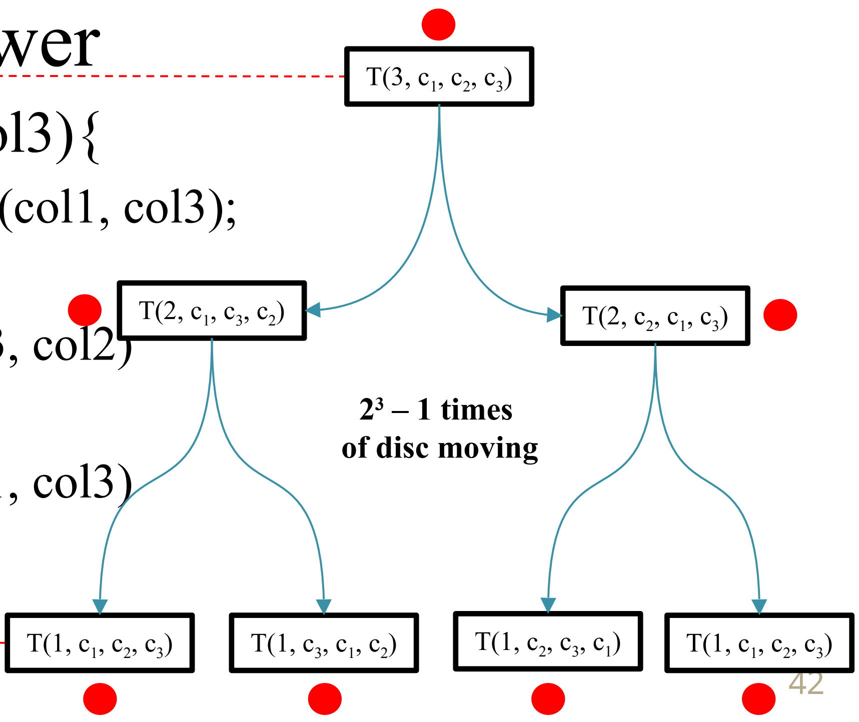


EXTENDED PROBLEMS

- The algorithm's complexity
  - Space: memory is computed with the depth of the recursive tree (exclude nodes with the same level)
  - Time: count the operations in the recursive tree
- Example: Ha-Noi tower

```
Tower(n, col1, col2, col3){
    If(n = 1) Then MoveDisc(col1, col3);
    Else{
        Tower(n – 1, col1, col3, col2)
        MoveDisc(col1, col3);
        Tower(n – 1, col2, col1, col3)
    }
}
```

Depth = 3 (storage unit)



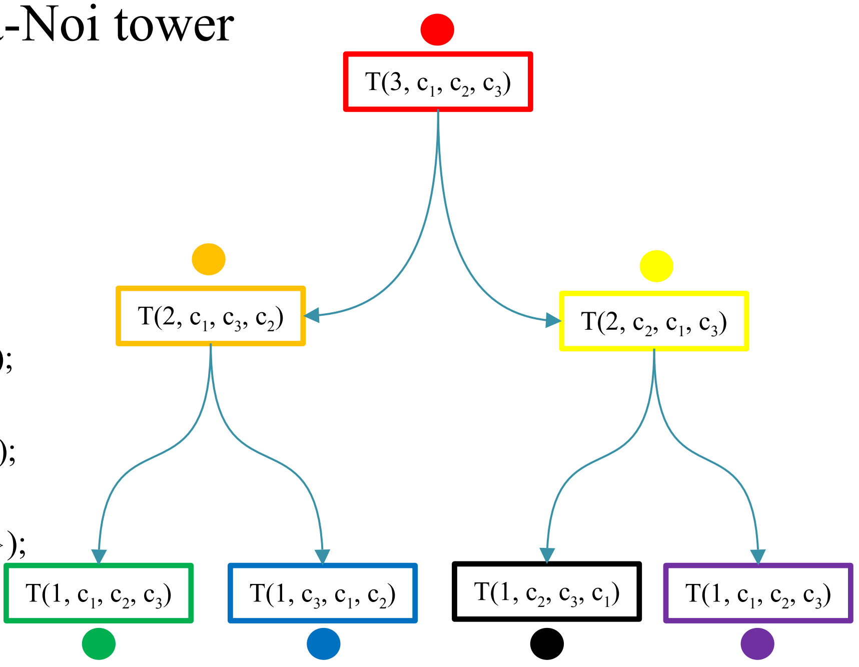
EXTENDED PROBLEMS

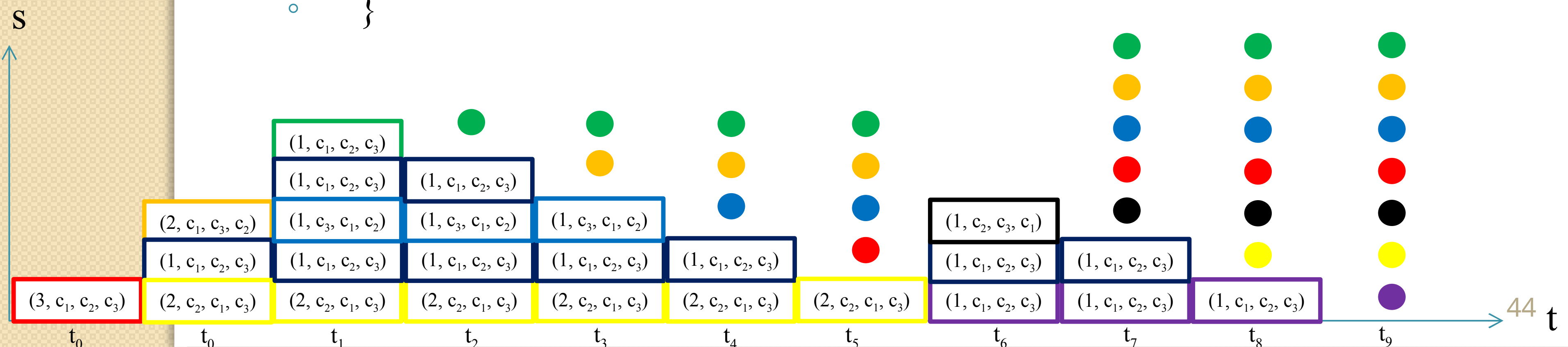
- Replacing recursion
  - Recursion algorithm consume memory and time very much
  - In some case, recursion algorithm is simple and easy-to-understand
  - Completely convert a recursive algorithm to non-recursive one with loop and stack
  - Using stack is more efficient time and space than recursive algorithm

EXTENDED PROBLEMS

- Non-recursive algorithm for Ha-Noi tower

```
Tower(n, col1, col2, col3){
    Stack s;
    Push(s, kq ⟸ {n , col1, col2, col3});
    Do{
        kq ⟸ pop(s);
        If(kq.n = 1) MoveDisc(kq.col1, kq.col3);
        Else{
            Push(s, kq ⟸ {n – 1, col2, col1, col3});
            Push(s, kq ⟸ {1, col1, col2, col3});
            Push(s, kq ⟸ {n - 1}, col1, col3, col2});
        }
    }Until(!empty(s));
}
```





EXTENDED PROBLEMS

- The correctness of recursive algorithm
  - Prove the output of recursive program is right
  - Use induction to prove
    - Example $n! = 1 (n = 0)$ and $n! = n*(n-1)! \ (n > 0)$
    - Program
      - GT(n){
        - if n = 0 then kq = 1
        - else kq = n * GT(n – 1)
      - }
    - Proof: GT(n) = n!
      - If $n = 0$ then kq ⩦ 1 (Definition)
      - Assume the program is right with $n = k$, so GT(k) = k!
      - Need to prove GT($k + 1$) = $(k + 1)!$
        - Easy GT($k + 1$) = $(k + 1)*$GT(k)
        - We have GT(k) = k!
        - So GT($k + 1$) = $(k + 1)*k! = (k + 1)!$ (complete the proof)

EXTENDED PROBLEMS

- The correctness of recursive algorithm
  - Prove the output of recursive program is right
  - Use induction to prove
    - Example: GCD(a, b) (a, $b > 0$)
    - Program

```
GCD(a, b){
    if a = b then d = a // kq = b
    else
        if  a > b then d = GCD(a – b, b)
        else d = GCD(a, b – a)
}
```

- Proof GCD(a, b) = d ($d|a \wedge d|b \wedge \nexists c$: $c|a \wedge c|b \wedge d<c$)
  - If $a = b$ then d = a (Right)
  - If $a \neq b$:
    - If $a > b$: because $d \mid a$ & $d \mid b$ so $(a/d) – (b/d) = (a – b)/d \in \mathbb{N} \Rightarrow d \mid a – b$. So d is the common divisor of $a – b$ and b
    - If $b > a$: because $d \mid a$ & $d \mid b$ so $(b/d) – (a/d) = (b – a)/d \in \mathbb{N} \Rightarrow d \mid b – a$. So d is the common divisor of $b – a$ and a
    - So the process continues: the bigger subtracts the smaller until $a – b = 0$ or $b – a = 0$. When $a = b$ then the algorithm stops