Chapter 10
Recursion

Problem Solving and Program Design in C
by Jeri R. Hanly and Elliot B. Koffman
Pei-yih Ting

"To Iterate is Human, to Recurse, Divine"
-- *L. Peter Deutsch*

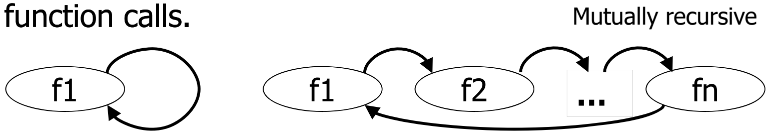
" To err is human; to really foul things up requires a computer "
-- *Bill Vaughan*

Outline

- Nature of Recursion
- Tracing a Recursive Function
- Recursive Mathematical Functions
- Case Study: Recursive Selection Sort
- A Classic Case Study: Towers of Hanoi
- Common Programming Errors

Recursive Function

- A **recursive function** is
  - a kind of function that calls itself, or
  - a function that is part of a cycle in the sequence of function calls.



- An <u>alternative to iteration</u> (looping) Tail recursive equivalent
  - A recursive solution is often **less efficient** than an iterative solution in terms of computer time due to the overhead for the extra function calls.
  - **More expressive (easier to write)**
  - Design is close to **mathematical induction**
  - Design is an application of divide and conquer

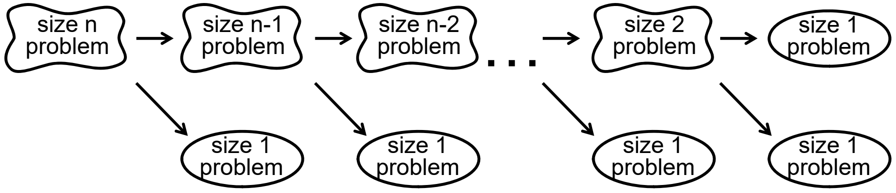
Nature of Recursion

- **Characteristics** of recursive solutions
  - One or more **simple cases** of the problem have straightforward, non-recursive solutions
  - The **other cases can be redefined** in terms of problems that are closer to the simple cases
  - By applying this redefinition process every time the recursive function is called, **eventually the problem is reduced entirely to simple cases**, which are relatively easy to solve
- **Basic algorithm**

```
if this is a simple case
   solve it
else
   redefine the problem using solutions to simpler
       problems
```

Splitting a Problem

- If the problem of **size 1** can be solved easily (i.e., the simple case).
- If the problem of **size n** can be splitted easily into a problem of **size 1** and another problem of **size n-1**.

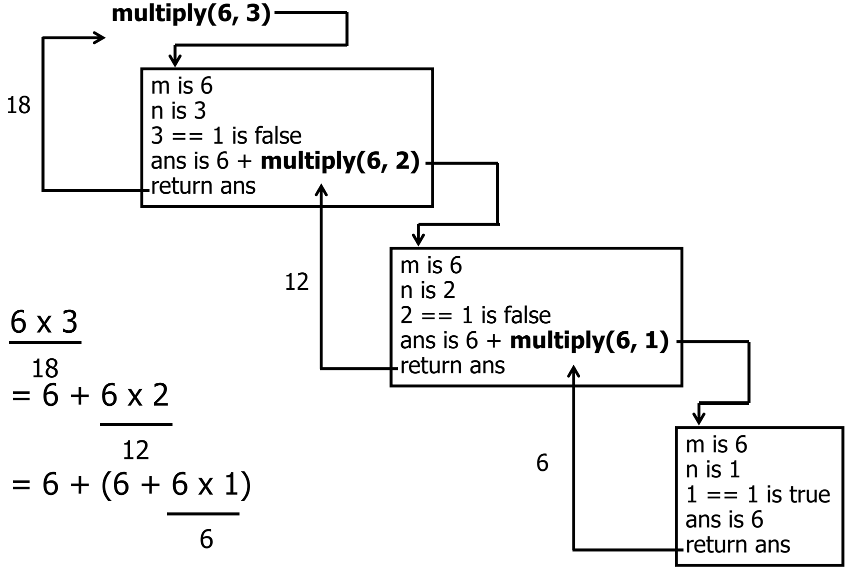


Splitting a Problem (cont’d)

- An illustrative example: **multiplication by addition**

```
/*
 *  Performs integer multiplication using + operator.
 *  Pre:   m and n are defined and n > 0
 *  Post:  returns m * n
 */
int
multiply(int m, int n)
{
    int ans;

    if (n == 1)
        ans = m;                                  /* simple case */
    else
        ans = m + multiply(m, n  - 1);  /* recursive step */

    return (ans);
}
```

Tracing a Recursive Function



multiply(6, 3)

18

m is 6
n is 3
3 == 1 is false
ans is 6 + **multiply(6, 2)**
return ans

12

m is 6
n is 2
2 == 1 is false
ans is 6 + **multiply(6, 1)**
return ans

6

m is 6
n is 1
1 == 1 is true
ans is 6
return ans

$$\underbrace{6 \times 3}_{18}$$

$$= 6 + \underbrace{6 \times 2}_{12}$$

$$= 6 + (6 + \underbrace{6 \times 1}_{6})$$

Terminating Condition

- A recursive function always contains one or more **terminating conditions**.
  - A condition when a recursive function is processing a simple case instead of processing recursion.
- Without suitable terminating conditions, the recursive function may run forever.

e.g., in the previous multiply function, the if statement “**if (n == 1) ...**” is the terminating condition.

Recursive Character Counting

- Count the number of occurrences of a given character in a string.

Figure 10.4

e.g., the number of 's' in "Mississippi" is 4.

Mississippi sassafras

If I could just get <u>someone</u> to count the s's in this list,

then the number of s's is either that number or 1 more, depending on whether the **first letter** is an 's'.

Character Counting (cont'd)

```
/*
 *  Count the number of occurrences of character ch in string str
 */
int
count(char ch, const char *str)
{
    int ans;

    if (str[0] == '\0')    /*  simple case  */
        ans = 0;
    else                   /*  redefine problem using recursion */
        if (ch == str[0])  /*  first character must be counted    */
            ans = 1 + count(ch, &str[1]);
        else               /*  first character is not counted       */
            ans = count(ch, &str[1]);

    return (ans);
}
```

```
char buf[] = "Mississippi";
count('s', buf);
```

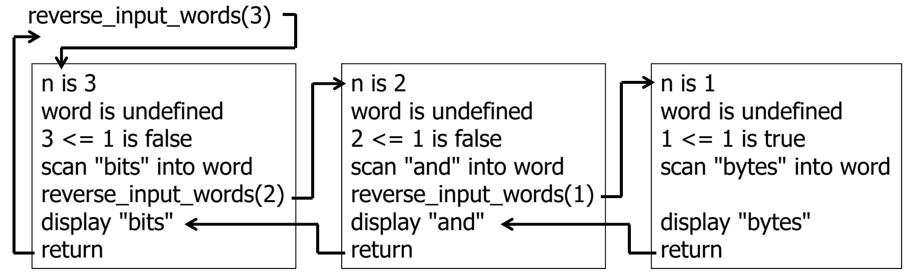
Reverse Input Words

```
/*
 *  Take n words as input and print them in reverse order on separate lines.
 *  Pre: n > 0
 */
void
reverse_input_words(int n)
{
    char word[WORDSIZ];  /*  local variable for storing one word */

    if (n <= 1) {   /* simple case: just one word to get and print */
        scanf("%s", word);
        printf("%s\n", word);
    } else {  /* get this word; get and print the rest of the words in
                 reverse order; then print this word  */
        scanf("%s", word);
        reverse_input_words(n - 1);
        printf("%s\n", word);
    }
}
```

The last scanned word is first printed.

The scanned word will not be printed until the recursion finishes.

Trace of Reverse Input Words



reverse_input_words(3)

n is 3
word is undefined
3 <= 1 is false
scan "bits" into word
reverse_input_words(2)
display "bits"
return

n is 2
word is undefined
2 <= 1 is false
scan "and" into word
reverse_input_words(1)
display "and"
return

n is 1
word is undefined
1 <= 1 is true
scan "bytes" into word

display "bytes"
return

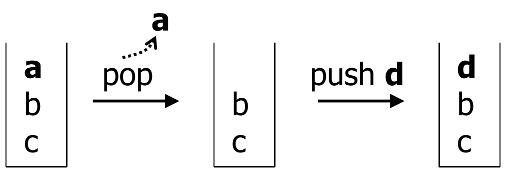
Sequence of Events

Input:
bits
and
bytes
Output:
bytes
and
bits

Call reverse_input_words with n equal to 3.
- Scan the first word ("bits") into word.
- Call reverse_input_words with n equal to 2.
  - Scan the second word ("and") into word.
  - Call reverse_input_words with n equal to 1.
    - Scan the third word ("bytes") into word.
    - Display the third word ("bytes").
    - Return from third call.
  - Display the second word ("and").
  - Return from second call.
- Display the first word ("bits").
- Return from original call.

How C Maintains Recursive Steps

- C keeps track of the values of variables and parameters by the **stack** data structure.
  - Recall that stack is a data structure where the last item added is the first item being processed. (LIFO)
  - There are two operations (push and pop) associated with stack.



Execution of Recursive Function

- Each time a function is called, the **execution state** of the caller function (e.g., parameters, local variables, and return address) are pushed onto the **stack** as an **activation frame**.
- When the execution of the called function is finished, the execution is restored by popping out the execution state from the stack.
- This is sufficient to maintain the execution of a recursive function.
  - The execution states of each recursive function are stored and kept in order on the stack.

Trace a Recursive Function

- A recursive function is not easy to trace or debug.
  - If there are hundreds of recursive steps, it is not useful to set the breaking point or to trace step-by-step.
- A useful approach is inserting printing statements and then watching the output (the calling sequence, arguments, and results) to trace the recursive steps.
- **When** and **how** to trace recursive functions
  - During algorithm development, it is best to trace a specific case simply by **trusting any recursive call to return a correct value** based on the function purpose.

Trace a Recursive Function (cont'd)

- Below is a self-tracing version of function multiply as well as output generated by the call.

```
int multiply(int m, int n) {
   int ans;
   printf("Entering multiply with m = %d, n = %d\n", m, n);
   if (n == 1)
      ans = m;
   else
      ans = m + multiply(m, n - 1);
   printf("multiply(%d, %d) returning %d\n", m, n, ans);
   return (ans);
}
```

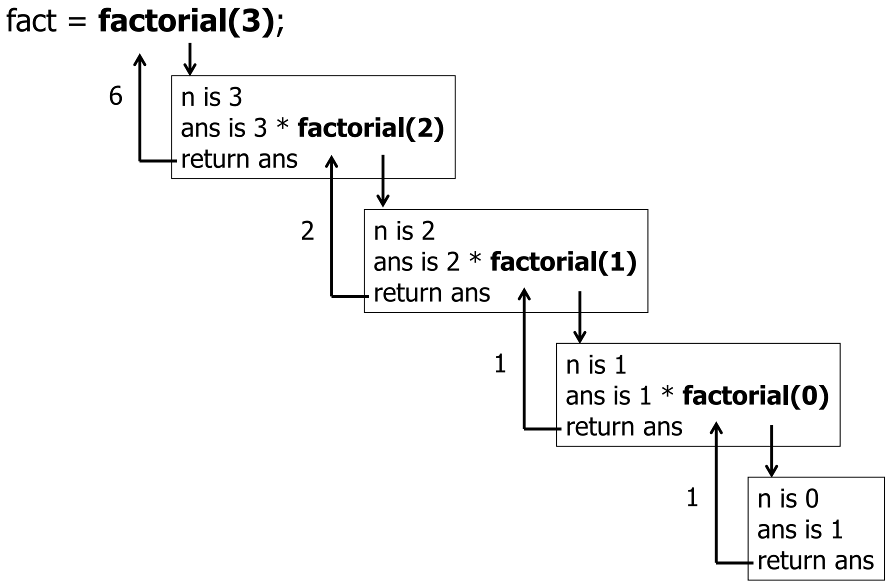
```
Entering multiply with m = 8, n = 3
Entering multiply with m = 8, n = 2
Entering multiply with m = 8, n = 1
multiply(8, 1) returning 8
multiply(8, 2) returning 16
multiply(8, 3) returning 24
```

Recursive Mathematical Functions

- Many mathematical functions can be defined and solved recursively, e.g. n!

```
/*
 *  Compute n! using a recursive definition
 *  Pre:  n >= 0
 */
int
factorial(int n)
{
    int ans;

    if (n == 0)
        ans = 1;
    else
        ans = n * factorial(n - 1);

    return (ans);
}
```

Trace of Recursive factorial



Iterative factorial

- The previous function can also be implemented by a for loop.
  - The iterative implementation is usually more efficient.

```
/*
 * Computes n! iteratively
 * Pre: n is greater than or equal to zero
 */
int factorial(int n)
{
    int i,                  /* local variables */
        product = 1;
    /* Compute the product n x (n-1) x (n-2) x ... x 2 x 1 */
    for  (i = n;  i > 1;  --i) {
        product = product * i;
    }
    /* Return function result */
    return (product);
}
```

Recursive fibonacci

- The Fibonacci numbers are a sequence of numbers that have many varied uses.
- The Fibonacci sequence is defined as
  - Fibonacci_1 is 1
  - Fibonacci_2 is 1
  - Fibonacci_n is Fibonacci_{n-2} + Fibonacci_{n-1}, for n>2

1, 1, 2, 3, 5, 8, 13, 21, 34, 55, …

Recursive fibonacci (cont’d)

```
/*
 *  Computes the nth Fibonacci number
 *  Pre: n > 0
 */
int
fibonacci(int n)
{
   int ans;

   if (n == 1  ||  n == 2)
      ans = 1;
   else
      ans = fibonacci(n - 2)  +  fibonacci(n - 1);

   return (ans);
}
```

Recursive gcd

- Euclidean algorithm for finding the greatest common divisor can be defined recursively
  - ❑ gcd(m,n) is n if n divides m evenly
  - ❑ gcd(m,n) is gcd(n, remainder of m divided by n) otherwise

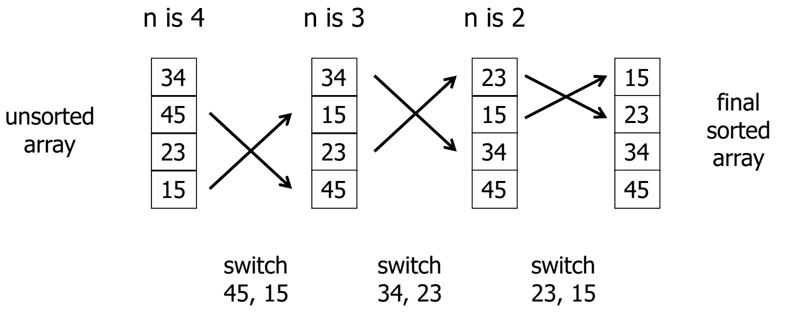
```
/*
 *  Find the greatest common divisor of m and n recursively
 *  Pre:  m and n are both > 0
 */
int gcd(int m, int n) {
   int ans;
   if (m % n == 0)
      ans = n;
   else
      ans = gcd(n, m % n);
   return (ans);
}
```

Recursive Selection Sort

Step 1: **Problem**

- ❑ Sort an array in ascending order using a selection sort.

n is the size of an unsorted array



Selection Sort (cont’d)

Step 3: **Design**

- ❑ Recursive algorithm for selection sort

```
1. if n is 1
      2. The array is sorted.
   else
      3. Place the largest array value in last array
         element
      4. Sort the subarray which excludes the last
         array element (array[0]..array[n-2])
```

Selection Sort (cont'd)

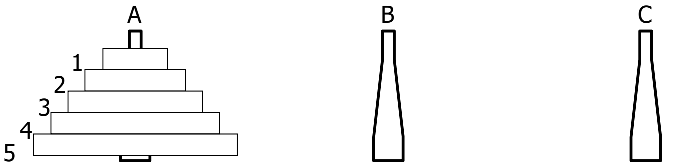
```
/*
 *  Finds the largest value in list array[0]..array[n-1] and exchanges it
 *                            with the value at array[n-1]
 *  Pre:  n > 0 and first n elements of array are defined
 *  Post: array[n-1] contains largest value
 */
void
place_largest(int array[], /* input/output - array in which to place largest  */
                      int n)          /* input - number of array elements to consider   */
{
    int temp,           /* temporary variable for exchange  */
         j,                  /* array subscript and loop control   */
         max_index;  /* index of largest so far                  */

    /*  Save subscript of largest array value in max_index    */
    max_index = n - 1;     /* assume last value is largest    */
    for  (j = n - 2;  j >= 0;  --j)
        if (array[j] > array[max_index])
            max_index = j;
```

Selection Sort (cont'd)

```
    /*  Unless last element is already the largest, exchange the largest and the last */
    if (max_index != n - 1) {
        temp = array[n - 1];
        array[n - 1] = array[max_index];
        array[max_index] = temp;
    }
}
/*
 *  Sorts n elements of an array of integers
 *  Pre:  n > 0 and first n elements of array are defined
 *  Post: array elements are in ascending order
 */
void
select_sort(int array[],  /* input/output - array to sort   */
                  int n)           /* input - number of array elements to sort   */
{
    if (n > 1) {
        place_largest(array, n);
        select_sort(array, n - 1);
    }
}
```

Towers of Hanoi

- The towers of Hanoi problem involves **moving a number of disks (in different sizes) from one tower (or called "peg") to another**.
  - The constraint is that the larger disk can never be placed on top of a smaller disk
  - Only one disk can be moved at each time
  - There are three towers



- Animation: http://www.mazeworks.com/hanoi/

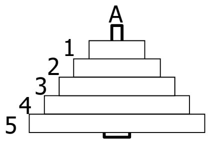
Tower of Hanoi (cont'd)

Step 2: **Analysis**

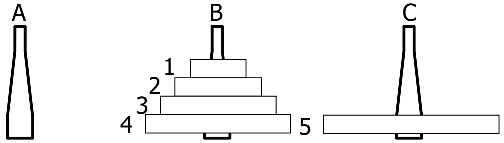
- Problem inputs
  - int n
  - char from_peg
  - char to_peg
  - char aux_peg
- Problem output
  - A list of individual disk moves

Tower of Hanoi (cont'd)

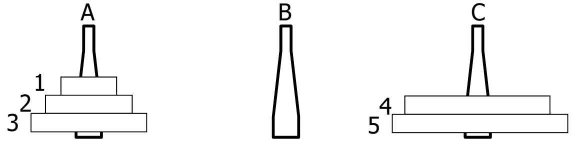
1. Move four disks from A to B.
2. Move disk 5 from A to C.
3. Move four disks from B to C.
    - 3.1 move three disks from B to A.
    - 3.2 Move disk 4 from B to C.
    - 3.3 Move three disks from A to C.



After Steps 1,2



After Steps 1,2,
3.1 and 3.2



Tower of Hanoi (cont'd)

```
/*
 *  Displays instructions for moving n disks from from_peg to to_peg using
 *  aux_peg as an auxiliary.  Disks are numbered 1 to n (smallest to
 *  largest). Instructions call for moving one disk at a time and never
 *  require placing a larger disk on top of a smaller one.
 */
void
tower(char from_peg,  /* input - characters naming          */
      char to_peg,    /*         the problem's              */
      char aux_peg,   /*         three pegs                 */
      int  n)         /* input - number of disks to move */
{
   if (n >= 1) {
      tower(from_peg, aux_peg, to_peg, n - 1);
      printf("Move disk %d from peg %c to peg %c\n", n, from_peg, to_peg);
      tower(aux_peg, to_peg, from_peg, n - 1);
   }
}
```

Output Generated

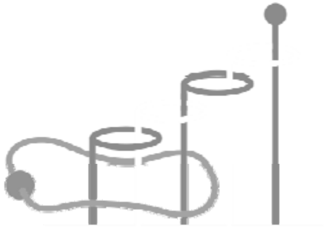
tower('A', 'C', 'B', 3);

Move top 3 disks from peg **A** to peg **C** using peg **B** as auxiliary peg

tower('A','B','C',2);	Move disk 1 from A to C	**tower('A','C','B',1);**
	Move disk 2 from A to B	**tower('A','B','C',1);**
	Move disk 1 from C to B	**tower('C','B','A',1);**
tower('A','C','B',1);	Move disk 3 from A to C	
tower('B','C','A',2);	Move disk 1 from B to A	**tower('B','A','C',1);**
	Move disk 2 from B to C	**tower('B','C','A',1);**
	Move disk 1 from A to C	**tower('A','C','B',1);**

Other Example

➢ 九連環



Iterative versus Recursive

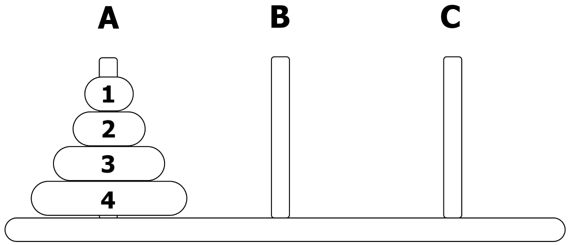
- **Recursive**
  - Requires **more time and space** because of extra function calls (not a problem for modern computer)
  - Much **easier to read and understand**
  - For researchers developing solutions to complex problems that are at the frontiers of their research areas, the benefits gained from **increased clarity far outweigh** the extra cost in **time and memory** of running a recursive program

---

Common Programming Errors

- A recursive function may **not terminate properly**.
  - A run-time error message noting **stack overflow** or an **access violation** is an indicator that a recursive function is not terminating
- Be aware that it is critical that every path through a non-void function leads to a return statement
- The recopying of **large arrays or other data structures** quickly consumes all available memory
  - cl /Ge } enable stack checks in VC
  - #pragma check_stack(on) }
  - cl /F10000000 self definition of stack size (bytes)
  - #pragma comment(linker, "/stack:xxx /heap:yyy")
  - -Wl,-stack,50000 for dev C++ linker

Hanoi Tower Animation

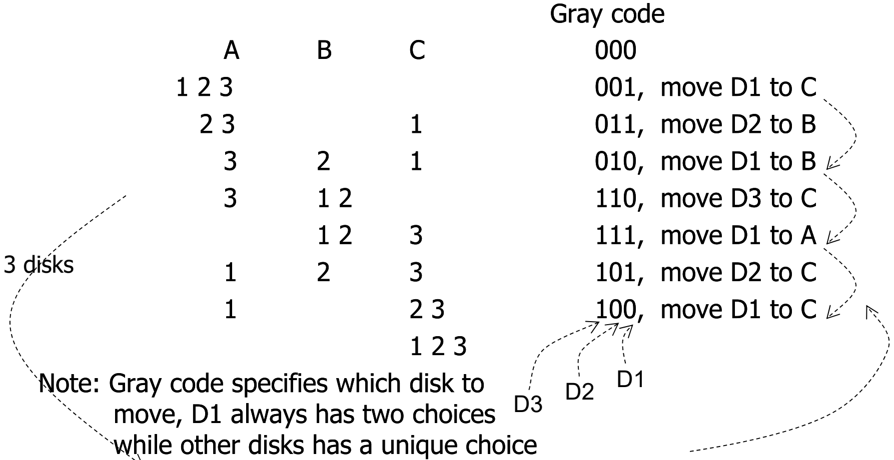


程式輸出

```
1, A -> B    1, C -> A    1, B -> C    1, A -> B
2, A -> C    2, C -> B    2, B -> A    2, A -> C
1, B -> C    1, A -> B    1, C -> A    1, B -> C
3, A -> B    4, A -> C    3, B -> C
```

Iterative Hanoi Tower

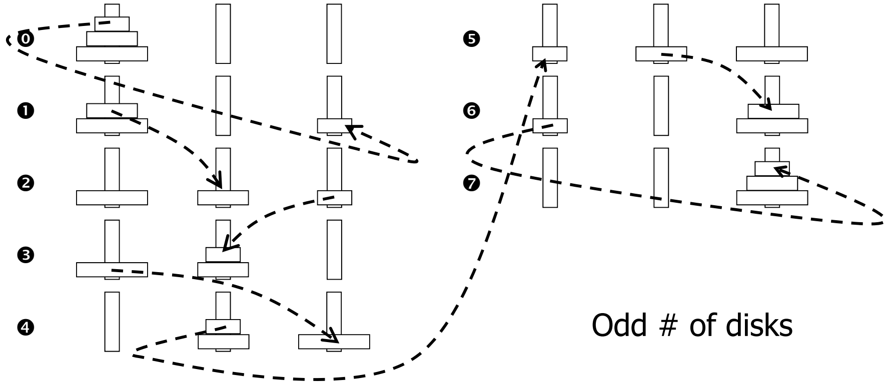
- Hanoi Tower



A	B	C	Gray code
			000
1 2 3			001, move D1 to C
2 3		1	011, move D2 to B
3	2	1	010, move D1 to B
3	1 2		110, move D3 to C
	1 2	3	111, move D1 to A
1	2	3	101, move D2 to C
1		2 3	100, move D1 to C
		1 2 3	

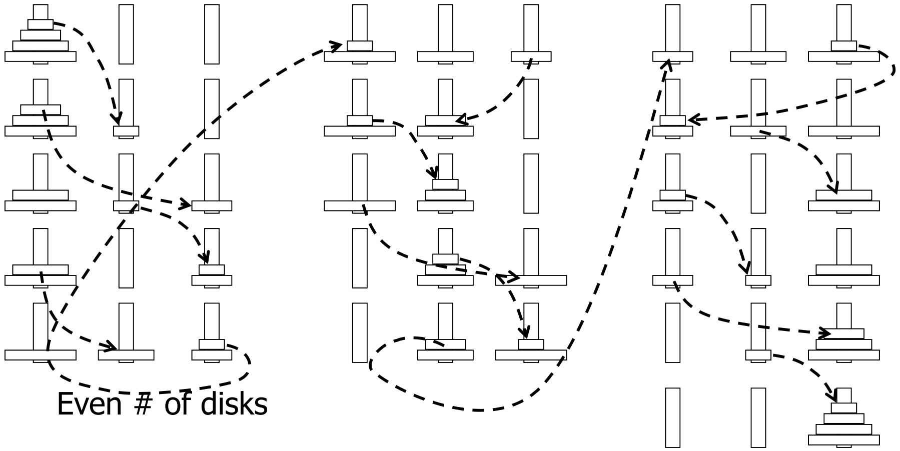
Note: Gray code specifies which disk to move, D1 always has two choices while other disks has a unique choice
for odd # of disks, D1 uses the sequence C B A C B A …
for even # of disks, D1 uses the sequence B C A B C A …

Iterative Hanoi Tower



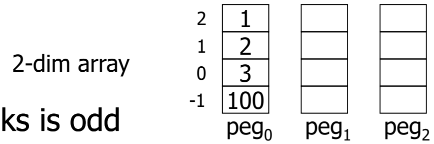
- **Alternate** moves between the smallest disk and a non-smallest disk.
- **For the smallest**: always move to the right (# of pieces is even), rotate if necessary; always move to the **left** (# of pieces is odd), rotate also
- **For the non-smallest**: there is only one possible legal move.

Iterative Hanoi Tower (cont'd)



- **Alternate** moves between the smallest disk and a non-smallest disk.
- **For the smallest**: always move to the **right** (# of pieces is even), rotate if necessary; always move to the left (# of pieces is odd), rotate also
- **For the non-smallest**: there is only one possible legal move.

Implementation



-1 mod 3 = 2
3 mod 3 = 0

-1 % 3 = -1
3 % 3 = 0

```
# of disks is odd
while not all disks in $peg_2$
    if $disk_1$ on $peg_i$, move $disk_1$ from $peg_i$ to $peg_{(i-1) \bmod 3}$
    find smaller $disk_j \neq disk_1$ on the top of $peg_i$ or $peg_{(i+1) \bmod 3}$,
    move $disk_j$ to the last peg
# of disks is even
while not all disks in $peg_2$
    find $disk_1$ on $peg_i$, move $disk_1$ from $peg_i$ to $peg_{(i+1) \bmod 3}$
    find smaller $disk_j \neq disk_1$ on the top of $peg_i$ or $peg_{(i-1) \bmod 3}$,
    move $disk_j$ to the last peg
```

Simpler rule for the moving directions for disk i, i=1, 2, 3, …, n:
if **n-i** is odd, move rightward, else move leftward

Quick Sort

- Ex. 9, 5, 12, 19, 2, 6, 4, 15, 8, 3, 7
- Goal: 2, 3, 4, 5, 6, 7, 8, 9, 12, 15, 19
- Algorithm:
  - ❑ Divide and Conquer
  - ❑ At each step, put an arbitrary element in its correct place and partition the numbers into two groups

    e.g. put 9 in its place

							9			

{5, 2, 6, 4, 8, 3, 7} {12, 19, 15}

  - ❑ Now we have two sort problems with smaller sizes
  - ❑ Question: how do we do the partitioning efficiently

Quick Sort (cont'd)

pivot
istart

9, 5, 12, 19, 2, 6, 4, 15, 8, 3, 7

front iend rear

步驟 1. while rear>istart &&
data[rear]>=data[pivot]
rear--

9, 5, 12, 19, 2, 6, 4, 15, 8, 3, 7
front rear

步驟 2. while front<iend &&
data[front]<data[pivot]
front++

9, 5, 12, 19, 2, 6, 4, 15, 8, 3, 7
front rear

步驟 3. if rear>front
交換 data[front]及data[rear]
front++, rear--

9, 5, 7, 19, 2, 6, 4, 15, 8, 3, 12
front rear

重複步驟 1 至步驟 3
直到 front > rear
(i.e. while (front <= rear) { … })

9, 5, 7, 3, 2, 6, 4, 15, 8, 19, 12

9, 5, 7, 3, 2, 6, 4, 8, 15, 19, 12

交換 data[pivot] 及 data[rear]
得到兩個長度比較短的排序問題

9, 5, 7, 3, 2, 6, 4, 8, 15, 19, 12

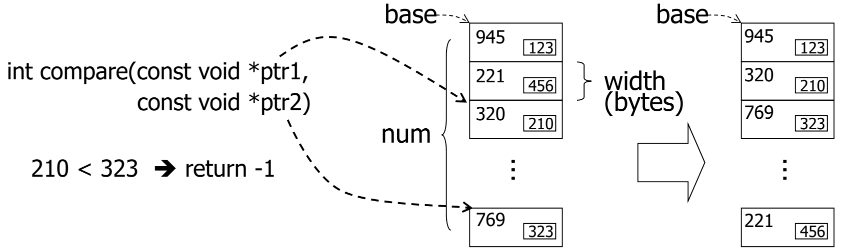
8, 5, 7, 3, 2, 6, 4, 9, 15, 19, 12

---

stdlib qsort()

```
#include <stdio.h>
#include <stdlib.h> // qsort()
int compare(const void *a, const void *b) {
    int *p1 = (int *) a, *p2 = (int *) b;
    return p1[1] - p2[1];
}
void main() {
    int data[][2] = {{945, 123}, {221, 456}, {320, 210}, {769, 323}};
    int i, ndata=sizeof(data)/sizeof(int)/2;
    qsort(data, ndata, 2*sizeof(int), compare);
    for (i=0; i<ndata; i++) printf("%d ", data[i][0]);
}
```

Result: 945 320 769 221



int compare(const void *ptr1,
const void *ptr2)

210 < 323 ➔ return -1

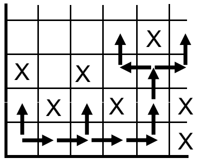
Maze

- Starting at the bottom-left corner, i.e. array index (8, 0), list any path through the maze that reaches the top-right corner, i.e. array index (0, 8). Only horizontal and vertical moves are allowed. You cannot go outside the board.
- DFS, recursion
- At each position, there are at most 3 directions that need to be tried, some of them is invalid.

end

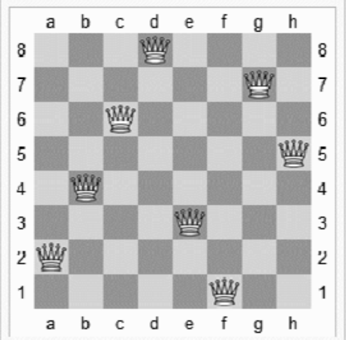
	0	1	2	3	4	5	6	7	8
0									
1		X	X	X		X		X	
2		X				X			
3				X			X	X	X
4	X	X	X		X		X		
5					X		X	X	
6	X		X					X	
7		X		X		X			
8						X		X	

start (8,0)



---

8 Queen Problem



One solution.

- Placing eight chess queens on an 8×8 chessboard so that none of them can capture any other using the standard chess queen's moves. Thus, a solution requires that no two queens share the same row, column, or diagonal.
- Brute force solution: DFS, recursion
- A fully brute force program need to search the $\mathbf{2^{8*8}}$ solution space.
- If consider both row and column constraints first, a solution must be a permutation. A brute force recursive program need to search the **8!** solution space.
- If both diagonal constraints are also considered, a queen must be placed at (x, y), which satisfies both x+y and x-y being unique.
- If rotations and reflections are counted as one, the 8 queen problem has **12** distinct solutions out of **92** unique solutions.
- There is a fast heuristic solution to n-queen problems. (see wiki)

Hanoi Tower Animation