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*

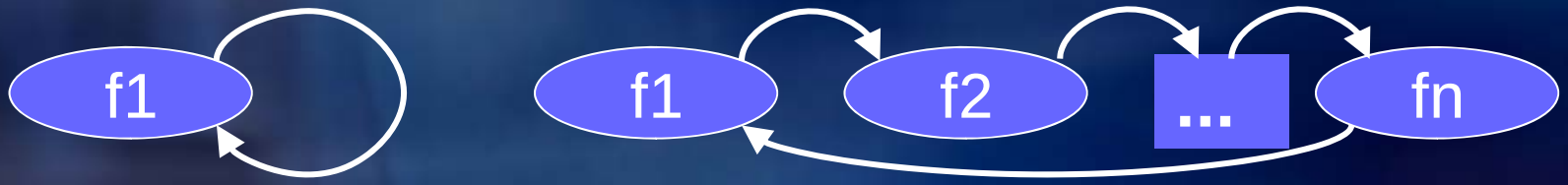
Outline

- 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.

Mutually recursive



- An alternative to iteration (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

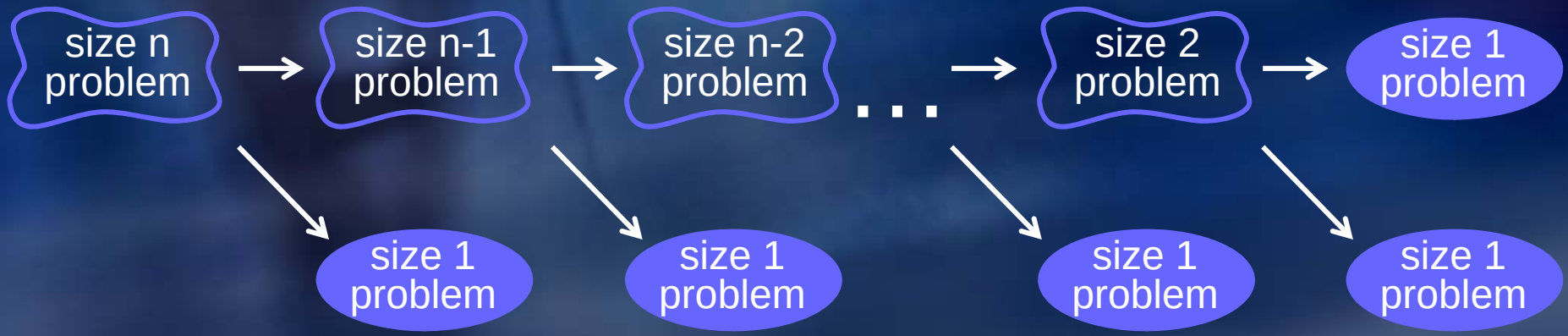
Nature of Recursion

- **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**.

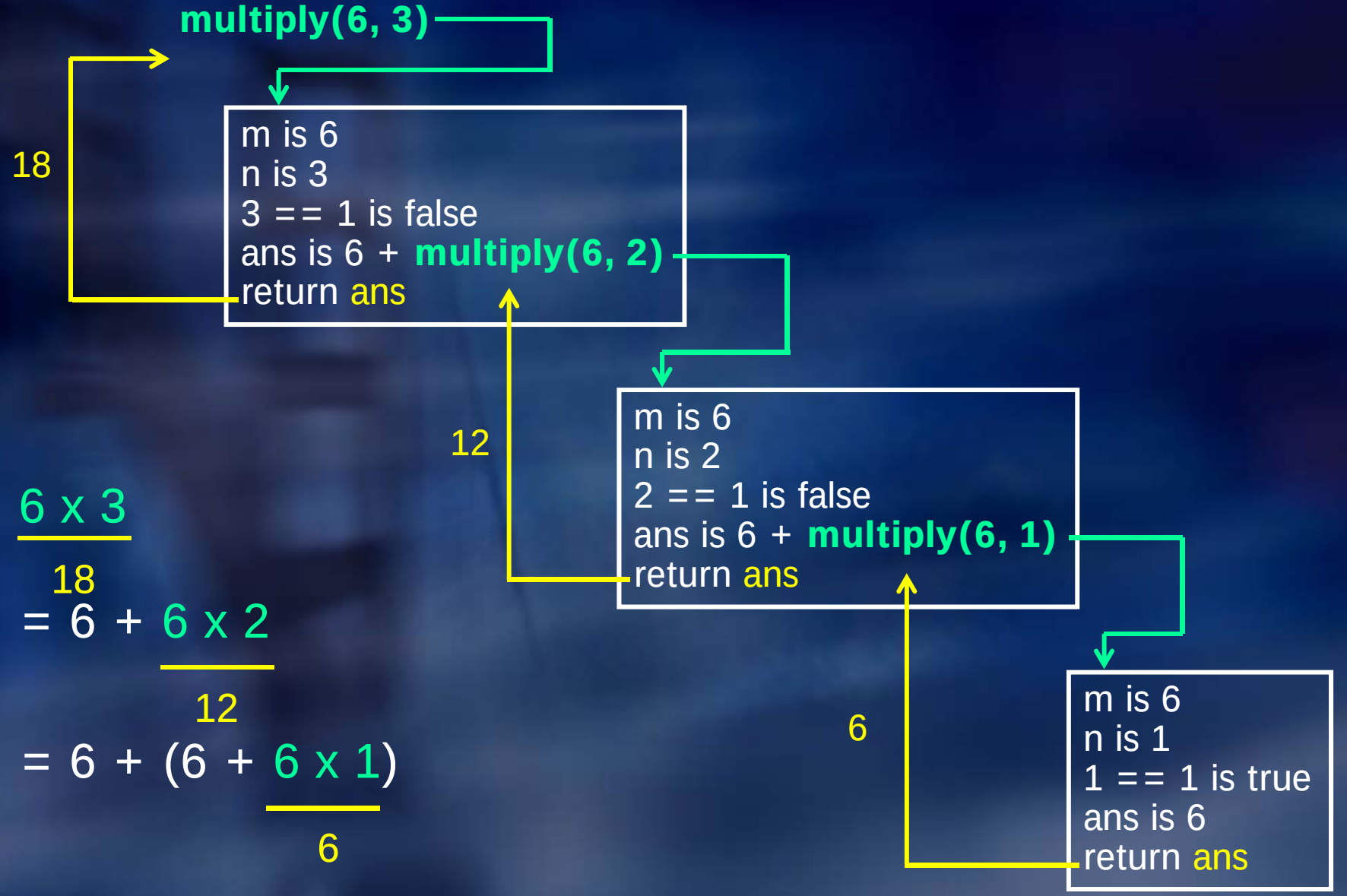


Splitting a Problem (cont'd)

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

```
01 /*
02  *  Performs integer multiplication using + operator.
03  *  Pre:   m and n are defined and n > 0
04  *  Post:  returns m * n
05  */
06 int
07 multiply(int m, int n)
08 {
09     int ans;
10
11     if (n == 1)
12         ans = m;                          /* simple case */
13     else
14         ans = m + multiply(m, n  - 1);  /* recursive step */
15
16     return (ans);
17 }
```

Tracing a Recursive Function



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.

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

Figure 10.4

M i s s i s s i p p i s a s s a f r a s

If I could just get someone 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)

```
01 /*
02  *  Count the number of occurrences of character ch in string str
03  */
04 int
05 count(char ch, const char *str)
06 {
07     int ans;
08
09     if (str[0] == '\0')     /*  simple case  */
10         ans = 0;
11     else                    /*  redefine problem using recursion */
12         if (ch == str[0])   /*  first character must be counted    */
13             ans = 1 + count(ch, &str[1]);
14         else                /*  first character is not counted      */
15             ans = count(ch, &str[1]);
16
17     return (ans);
18 }
```

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

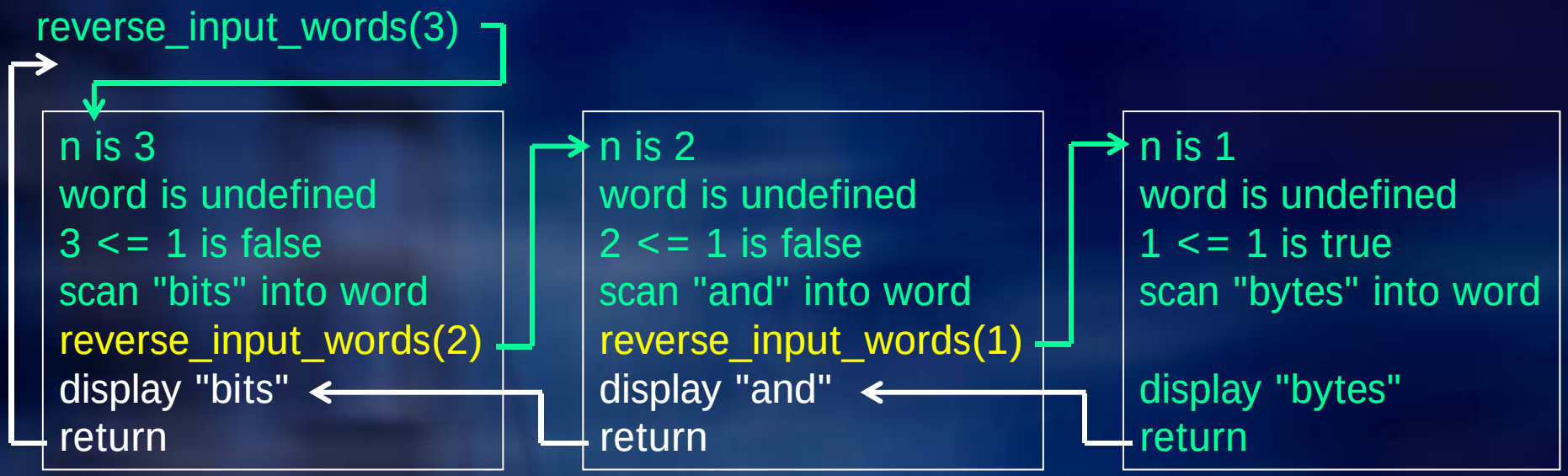
Reverse Input Words

```
01 /*
02  *  Take n words as input and print them in reverse order on separate lines.
03  *  Pre: n > 0
04  */
05 void
06 reverse_input_words(int n)
07 {
08     char word[WORDSIZ];  /*  local variable for storing one word */
09
10     if (n <= 1) {    /* simple case: just one word to get and print */
11         scanf("%s", word);
12         printf("%s\n", word);
13     } else {  /* get this word; get and print the rest of the words in
14                  reverse order; then print this word  */
15         scanf("%s", word);
16         reverse_input_words(n - 1);
17         printf("%s\n", word);
18     }
19 }
```

The last scanned word is first printed.

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

Trace of Reverse Input Words



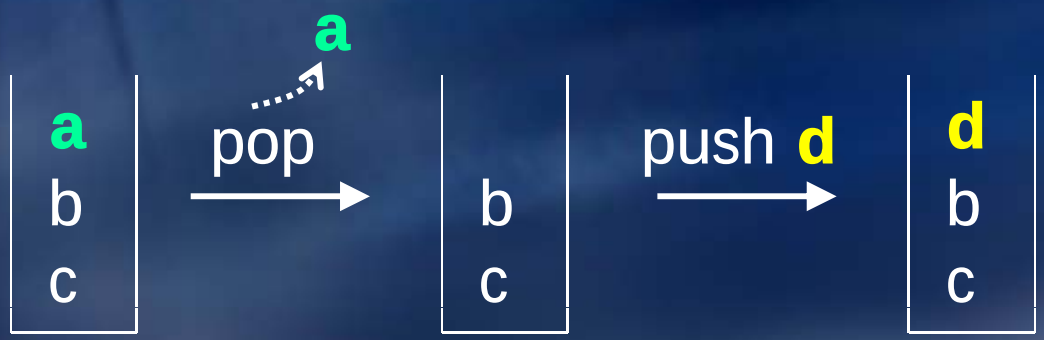
Sequence of Events

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.

```
01  int multiply(int m, int n) {
02      int ans;
03      printf("Entering multiply with m = %d, n = %d\n", m, n);
04      if (n == 1)
05          ans = m;
06      else
07          ans = m + multiply(m, n - 1);
08      printf("multiply(%d, %d) returning %d\n", m, n, ans);
09      return (ans);
10  }
```

```
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
```

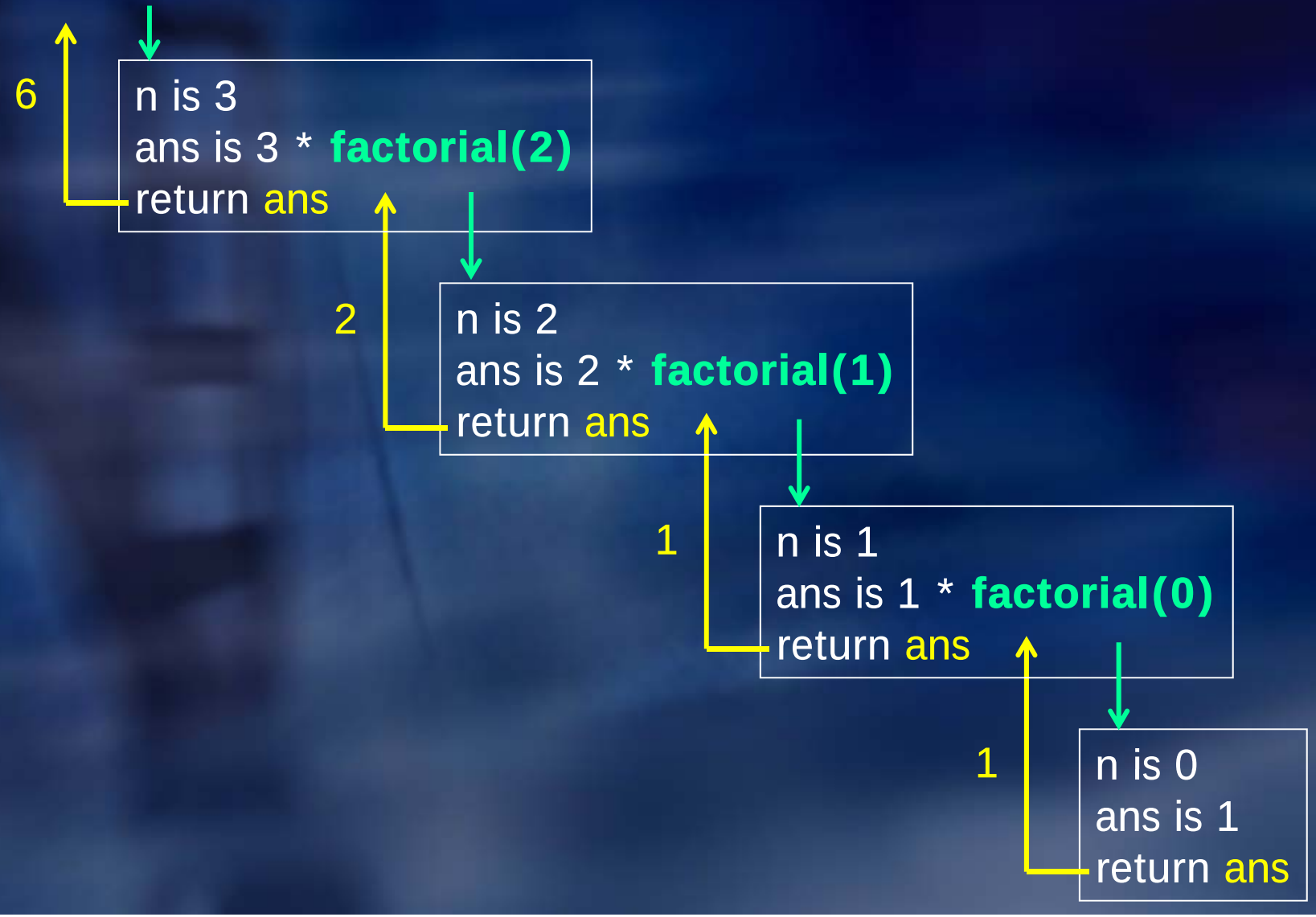
Recursive Mathematical Functions

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

```
01 /*
02  *  Compute n! using a recursive definition
03  *  Pre:  n >= 0
04  */
05 int
06 factorial(int n)
07 {
08     int ans;
09
10     if (n == 0)
11         ans = 1;
12     else
13         ans = n * factorial(n - 1);
14
15     return (ans);
16 }
```

Trace of Recursive factorial

fact = **factorial(3)**;



Iterative factorial

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

```
01 /*
02  * Computes n! iteratively
03  * Pre: n is greater than or equal to zero
04  */
05 int factorial(int n)
06 {
07     int i,                  /* local variables */
08         product = 1;
09     /* Compute the product n x (n-1) x (n-2) x ... x 2 x 1 */
10     for  (i = n;  i > 1;  --i) {
11         product = product * i;
12     }
13     /* Return function result */
14     return (product);
15 }
```

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)

```
01 /*
02  *  Computes the nth Fibonacci number
03  *  Pre: n > 0
04  */
05 int
06 fibonacci(int n)
07 {
08     int ans;
09
10     if (n == 1  ||  n == 2)
11         ans = 1;
12     else
13         ans = fibonacci(n - 2)  +  fibonacci(n - 1);
14
15     return (ans);
16 }
```

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

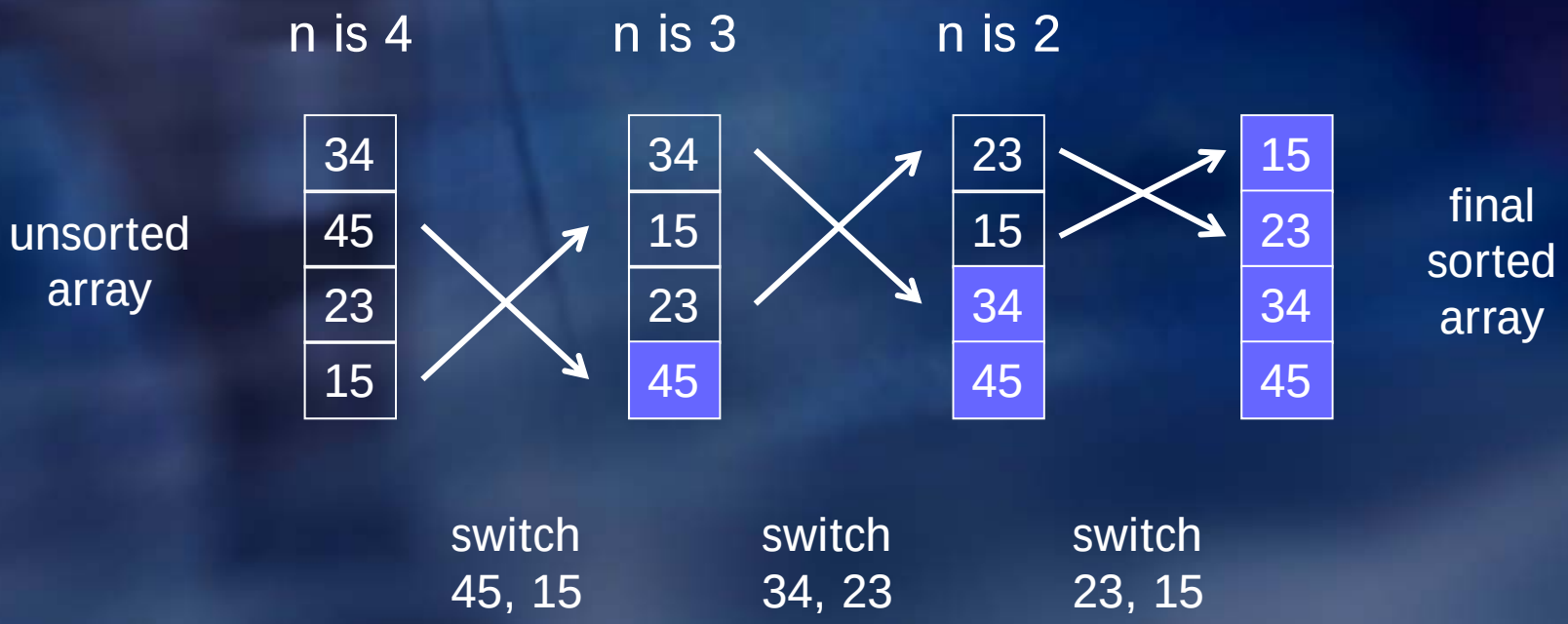
```
01 /*
02  *  Find the greatest common divisor of m and n recursively
03  *  Pre:  m and n are both > 0
04  */
05 int gcd(int m, int n) {
06     int ans;
07     if (m % n == 0)
08         ans = n;
09     else
10         ans = gcd(n, m % n);
11     return (ans);
12 }
```

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)

```
01 /*
02  *  Finds the largest value in list array[0]..array[n-1] and exchanges it
03  *                          with the value at array[n-1]
04  *  Pre:  n > 0 and first n elements of array are defined
05  *  Post: array[n-1] contains largest value
06  */
07 void
08 place_largest(int array[], /* input/output - array in which to place largest  */
09                             int n)          /* input - number of array elements to consider   */
10 {
11     int temp,            /* temporary variable for exchange  */
12         j,               /* array subscript and loop control   */
13         max_index;   /* index of largest so far                  */
14
15     /*  Save subscript of largest array value in max_index    */
16     max_index = n - 1;      /* assume last value is largest    */
17     for  (j = n - 2;  j >= 0;  --j)
18         if (array[j] > array[max_index])
19             max_index = j;
```

Selection Sort (cont'd)

```
21      /*  Unless last element is already the largest, exchange the largest and the last */
22      if (max_index != n - 1) {
23          temp = array[n - 1];
24          array[n - 1] = array[max_index];
25          array[max_index] = temp;
26      }
27  }
29  /*
30   *  Sorts n elements of an array of integers
31   *  Pre:  n > 0 and first n elements of array are defined
32   *  Post: array elements are in ascending order
33   */
34  void
35  select_sort(int array[],  /* input/output - array to sort    */
36                  int n)          /* input - number of array elements to sort    */
37  {
38      if (n > 1) {
39          place_largest(array, n);
40          select_sort(array, n - 1);
41      }
42  }
```

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/

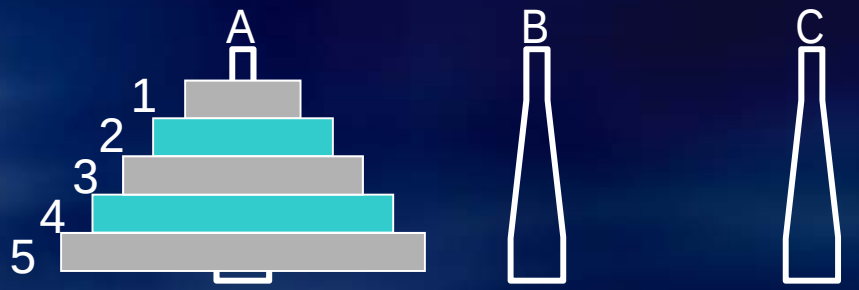
Tower of Hanoi (cont'd)

Step 2: **Analysis**

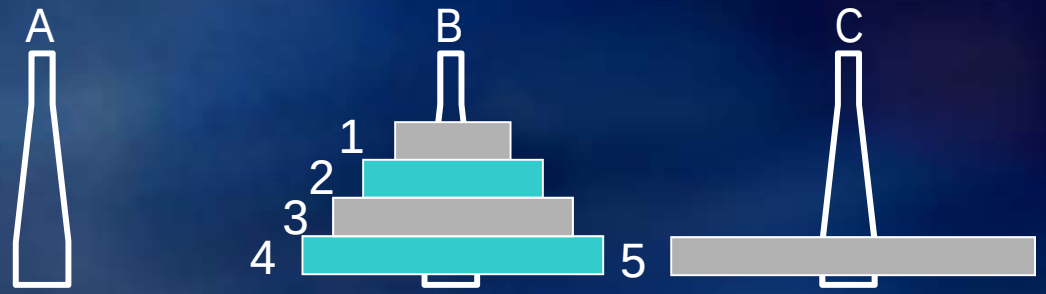
- Problem inputs
  - int n
  - char from_peg
  - char to_peg
  - char aux_peg
- Problem output
  - A list of individual disk moves

Tower of Hanoi (cont'd)

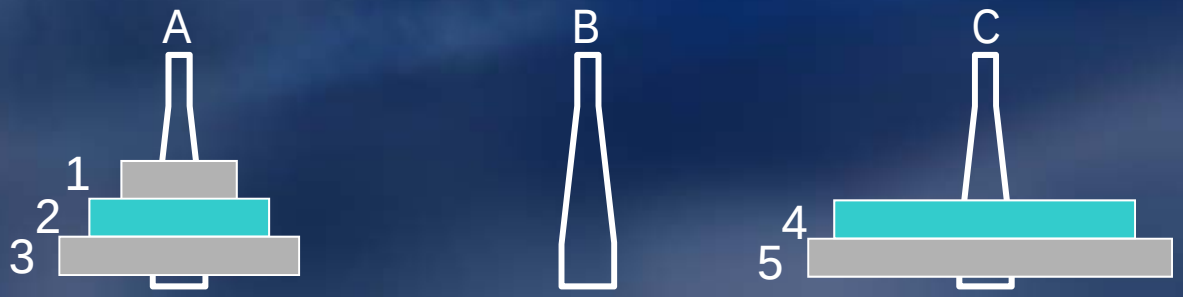
1. Move four disks from A to B.
2. Move disk 5 from A to C.
3. Move four disks from B to C.
   - 3.1 move three disks from B to A.
   - 3.2 Move disk 4 from B to C.
   - 3.3 Move three disks from A to C.



After Steps 1,2



After Steps 1,2,
3.1 and 3.2



Tower of Hanoi (cont'd)

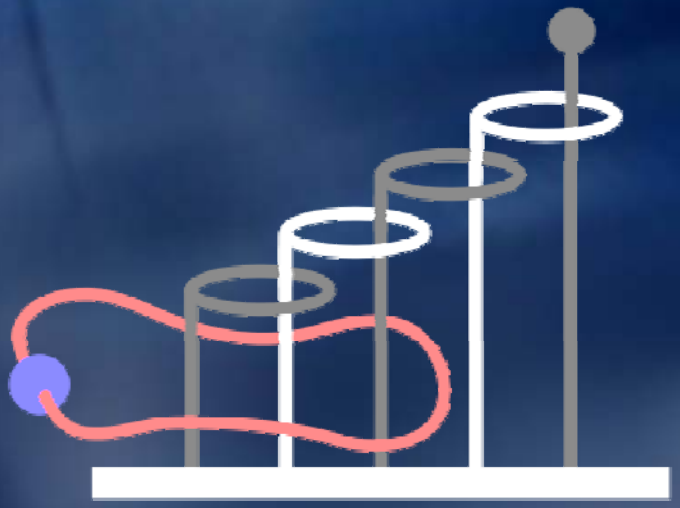
```
01 /*
02  *  Displays instructions for moving n disks from from_peg to to_peg using
03  *  aux_peg as an auxiliary.  Disks are numbered 1 to n (smallest to
04  *  largest). Instructions call for moving one disk at a time and never
05  *  require placing a larger disk on top of a smaller one.
06  */
07 void
08 tower(char from_peg,  /* input - characters naming          */
09           char to_peg,     /*           the problem's           */
10           char aux_peg,    /*           three pegs              */
11           int  n)          /* input - number of disks to move */
12 {
13     if (n == 1) {
14         printf("Move disk 1 from peg %c to peg %c\n", from_peg, to_peg);
15     } else {
16         tower(from_peg, aux_peg, to_peg, n - 1);
17         printf("Move disk %d from peg %c to peg %c\n", n, from_peg, to_peg);
18         tower(aux_peg, to_peg, from_peg, n - 1);
19     }
20 }
```

Output Generated

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

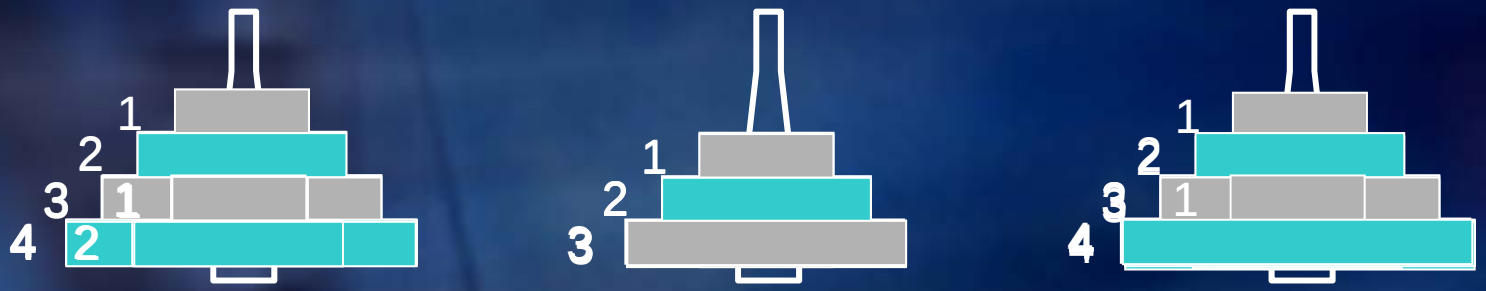
- **Recursive**
  - Requires **more time and space** because of extra function calls (not a problem for modern computer)
  - Much **easier to read and understand**
  - For researchers developing solutions to complex problems that are at the frontiers of their research areas, the benefits gained from **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
  - #pragma check_stack(on)

  } enable stack checks in VC
  - 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



Iterative Hanoi Tower

- Hanoi Tower

A	B	C	Gray code
1 2 3			000
2 3		1	001, move D1 to C
3	2	1	011, move D2 to B
3	1 2		010, move D1 to B
	1 2	3	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

3 disks

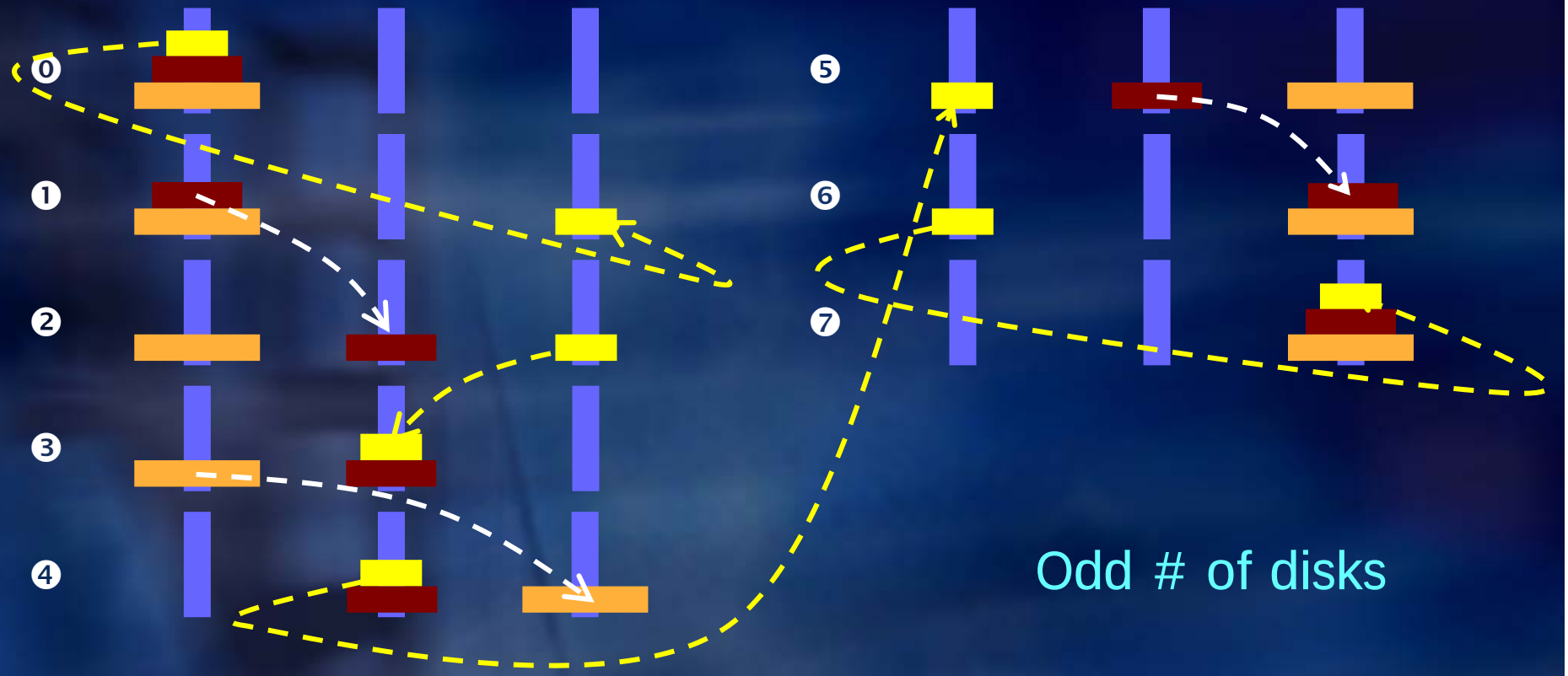
D3 D2 D1

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 ...

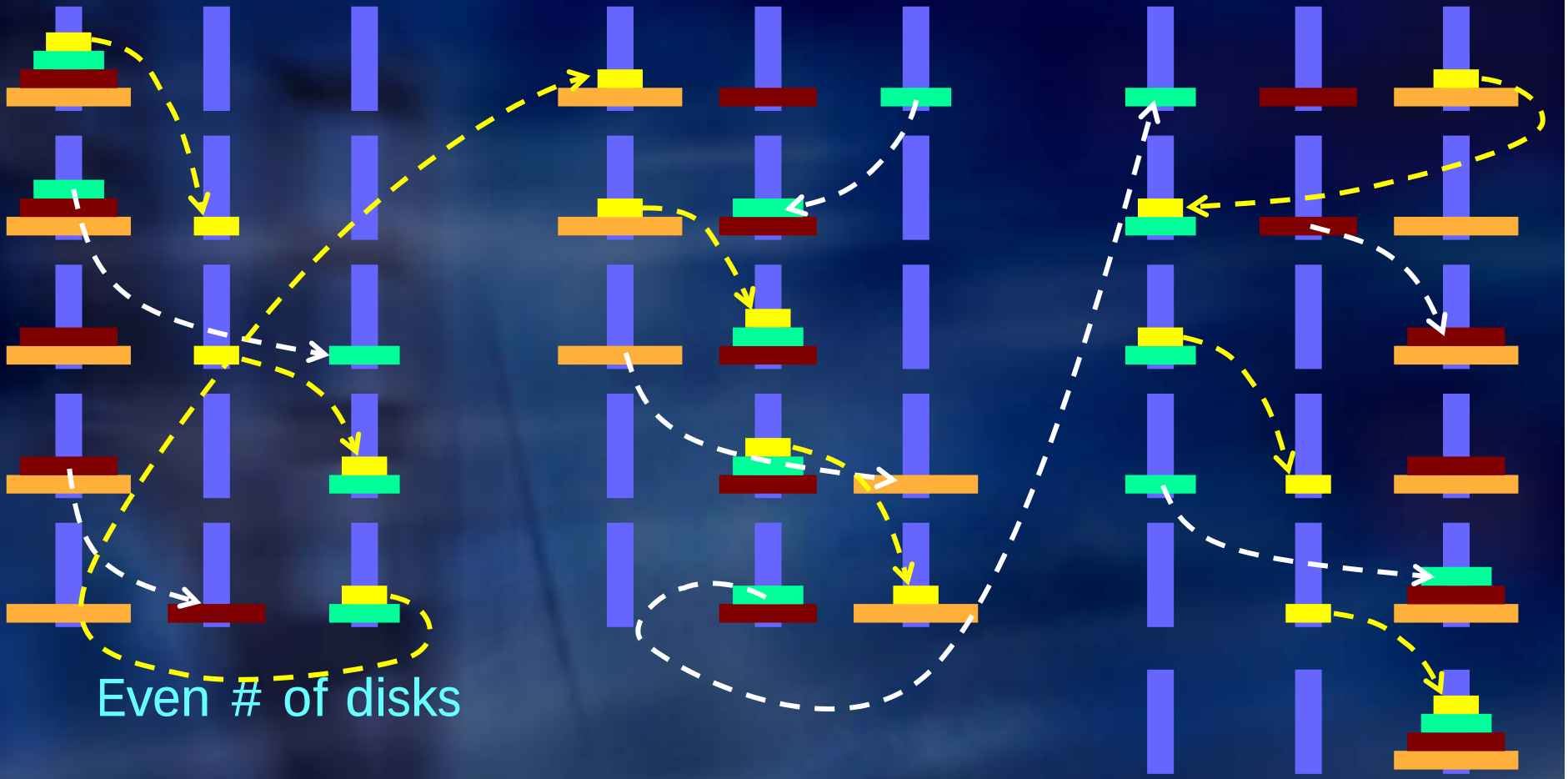
for even # of disks, D1 uses the sequence B C A B C A ...

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.

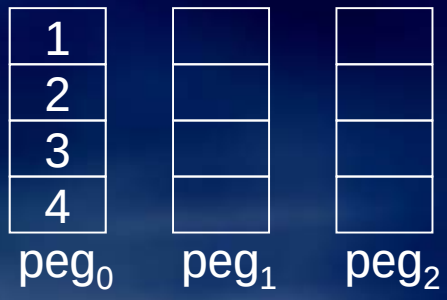
Iterative Hanoi Tower (cont'd)



- **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

2-dim array



-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

find $disk_1$ on peg_i, move $disk_1$ from peg_i to $peg_{(i-1) \bmod 3}$

find smallest $disk_j$ on the top of peg_i or $peg_{(i+1) \bmod 3}$,

move $disk_j$ to the remaining 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 smallest $disk_j$ on the top of peg_i or $peg_{(i-1) \bmod 3}$,

move $disk_j$ to the remaining 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

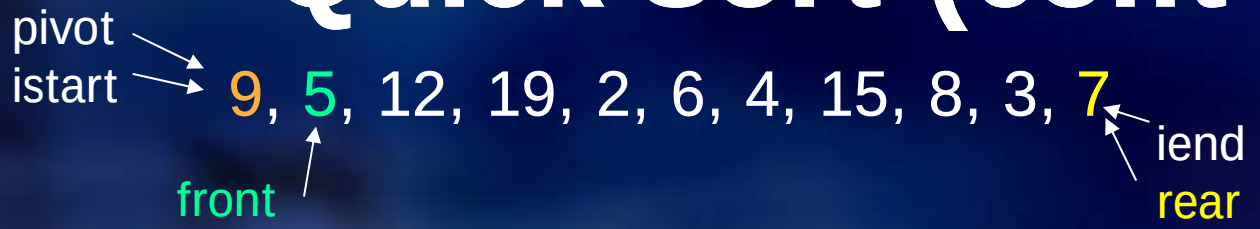
- Ex. 9, 5, 12, 19, 2, 6, 4, 15, 8, 3, 7
- Goal: 2, 3, 4, 5, 6, 7, 8, 9, 12, 15, 19
- Algorithm:
  - Divide and Conquer
  - At each step, put an arbitrary element in its correct place and partition the numbers into two groups

    e.g. put 9 in its place

{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)



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

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

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

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

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

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

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

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

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

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

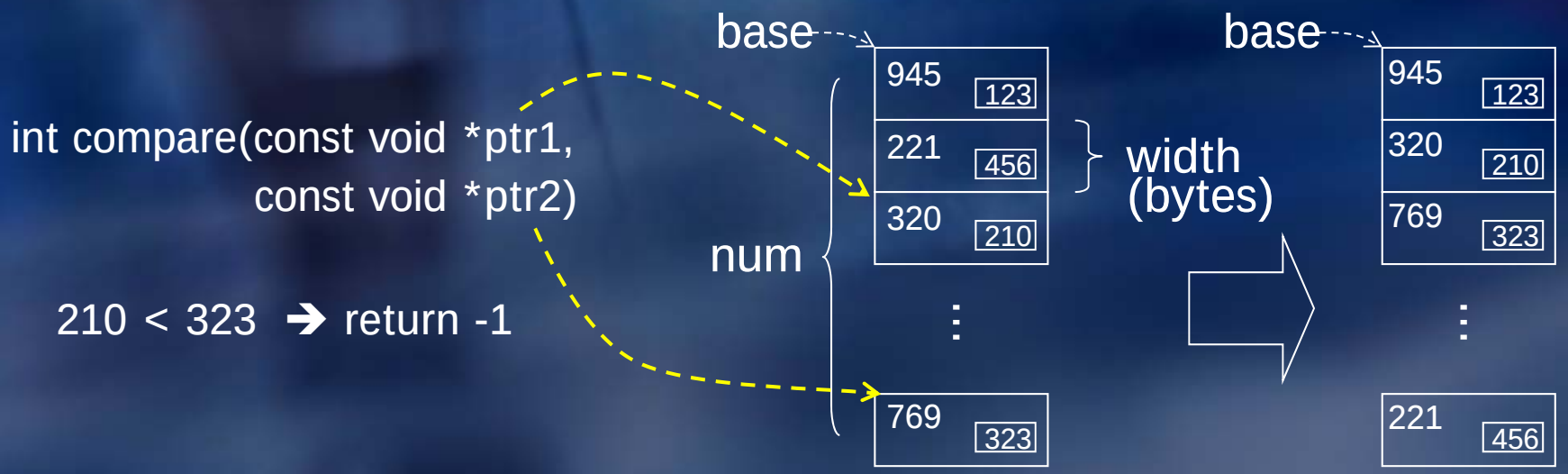
9, 5, 7, 3, 2, 6, 4, 8, 15, 19, 12

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

stdlib qsort()

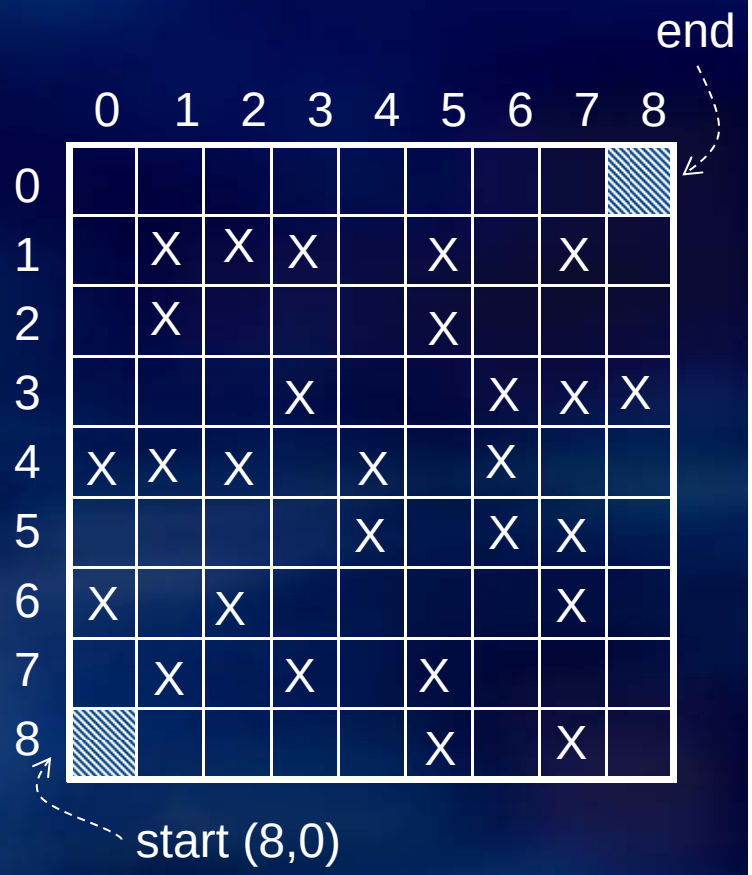
```
01 #include <stdio.h>
02 #include <stdlib.h> // qsort()
03 int compare(const void *a, const void *b) {
04     int *p1 = (int *) a, *p2 = (int *) b;
05     return p1[1] - p2[1];
06 }
07 void main() {
08     int data[][2] = {{945, 123}, {221, 456}, {320, 210}, {769, 323}};
09     int i, ndata=sizeof(data)/sizeof(int)/2;
10     qsort(data, ndata, 2*sizeof(int), compare);
11     for (i=0; i<ndata; i++) printf("%d ", data[i][0]);
12 }
```

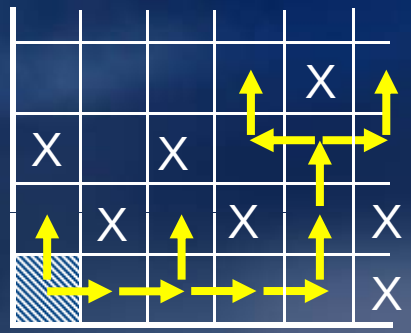
Result: 945 320 769 221



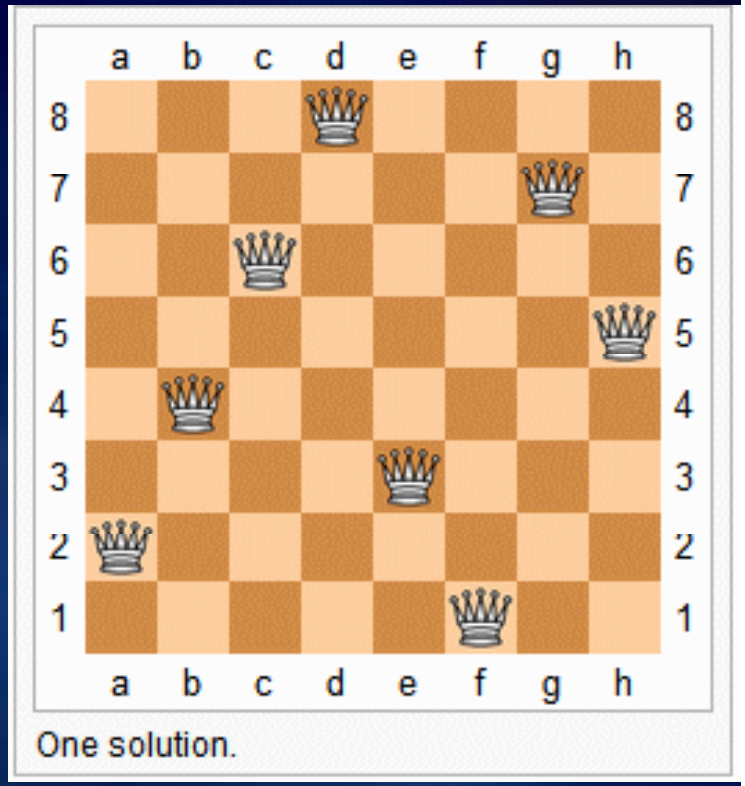
Maze

- 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.





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 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)