Recursion

Lecture 10

Instructor: Dr. Cong Pu, Ph.D.

cong.pu@okstate.edu

Adapted partially from Data Structures and Algorithms in Java, M.T. Goodrich, R.Tamassia and M. H. Goldwasser, Sixth Edition, Wiley; Data Structures and Algorithms in C++, Adam Drozdek, 4th Edition, Cengage Learning

Anatomy of a Recursive Call

- Analyze the recursive function and its behavior of recursion
  - e.g., a number *x* to a non-negative integer power *n*:

$$x^n = \begin{cases} 1 & \text{if } n = 0 \\ x \cdot x^{n-1} & \text{if } n > 0 \end{cases}$$

```
/* 102 */  double power (double x, unsigned int n) {
/* 103 */       if (n == 0)
/* 104 */          return 1.0;
                //else
/* 105 */       return x * power(x,n-1);
           }
```

- e.g., the calculation of x^4,
  - $x^4 = x \cdot x^3 = x \cdot (x \cdot x^2) = x \cdot (x \cdot (x \cdot x^1)) = x \cdot (x \cdot (x \cdot (x \cdot x^0))) = x \cdot (x \cdot (x \cdot (x \cdot 1))) = x \cdot (x \cdot (x \cdot (x))) = x \cdot (x \cdot (x \cdot x)) = x \cdot (x \cdot x \cdot x) = x \cdot x \cdot x \cdot x$
  - repeated application of the **inductive step** leads to the **anchor**

Anatomy of a Recursive Call (cont.)

- Produce the result of x^0, which is 1,
  - return this value to the previous call
- That call, which had been pending,
  - resume to calculate $x \cdot 1$, producing x
- Each succeeding return then takes the previous result
  - use it in turn to produce the final result

The sequence of recursive calls and returns,

call 1	$x^4 = x \cdot x^3$	$= x \cdot x \cdot x \cdot x$
call 2	$x^3 = x \cdot x^2$	$= x \cdot x \cdot x$
call 3	$x^2 = x \cdot x^1$	$= x \cdot x$
call 4	$x^1 = x \cdot x^0$	$= x \cdot 1$
call 5	$x^0 = 1$	

Anatomy of a Recursive Call (cont.)

- Alternatively,

call 1	`power(x,4)`
call 2	`power(x,3)`
call 3	`power(x,2)`
call 4	`power(x,1)`
call 5	`power(x,0)`
call 5	1
call 4	x
call 3	$x \cdot x$
call 2	$x \cdot x \cdot x$
call 1	$x \cdot x \cdot x \cdot x$

Anatomy of a Recursive Call (cont.)

- The system keeps track of *a sequence of calls* on the **runtime stack**,
  - store the **return address** of the function call
    - used to remember *where to resume execution* after the function has completed
  - e.g., `power( )` is called by the following statement in main():

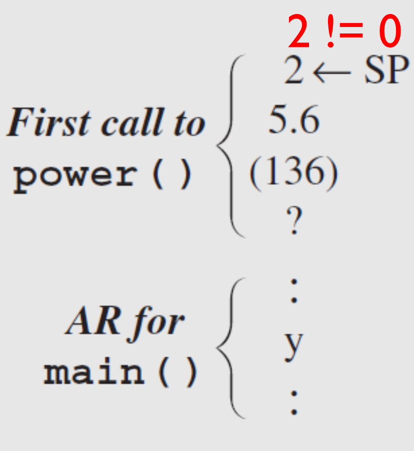
```
                 int main() {
/* 136 *                y = power(5.6,2);
                             ...
                 }
/* 102 */  double power (double x, unsigned int n) {
/* 103 */       if (n == 0)
/* 104 */          return 1.0;
                //else
/* 105 */       return x * power(x,n-1);
           }
```

Anatomy of a Recursive Call (cont.)

Key: SP Stack pointer
AR Activation record
? Location reserved for returned value

Changes to the run-time stack during execution of power(5.6,2)

Anatomy of a Recursive Call (cont.)

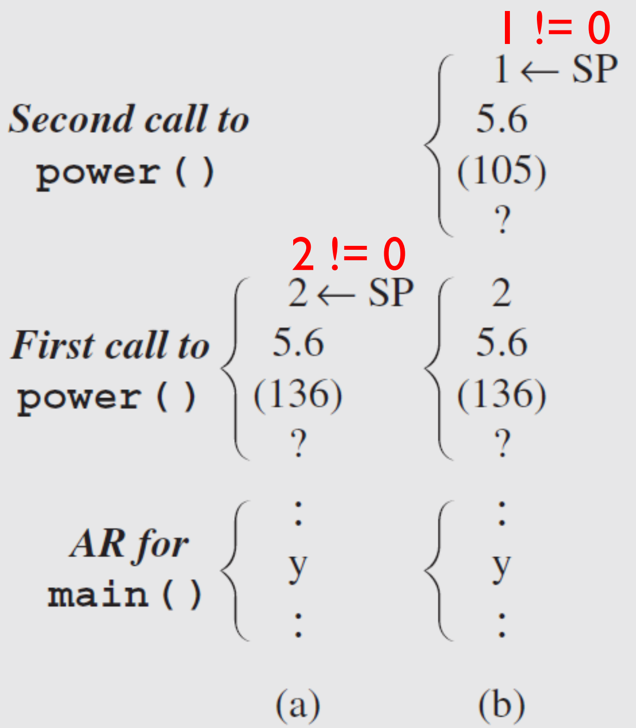


(a)

Changes to the run-time stack during execution of power(5.6,2)

Key:
SP Stack pointer
AR Activation record
? Location reserved for returned value

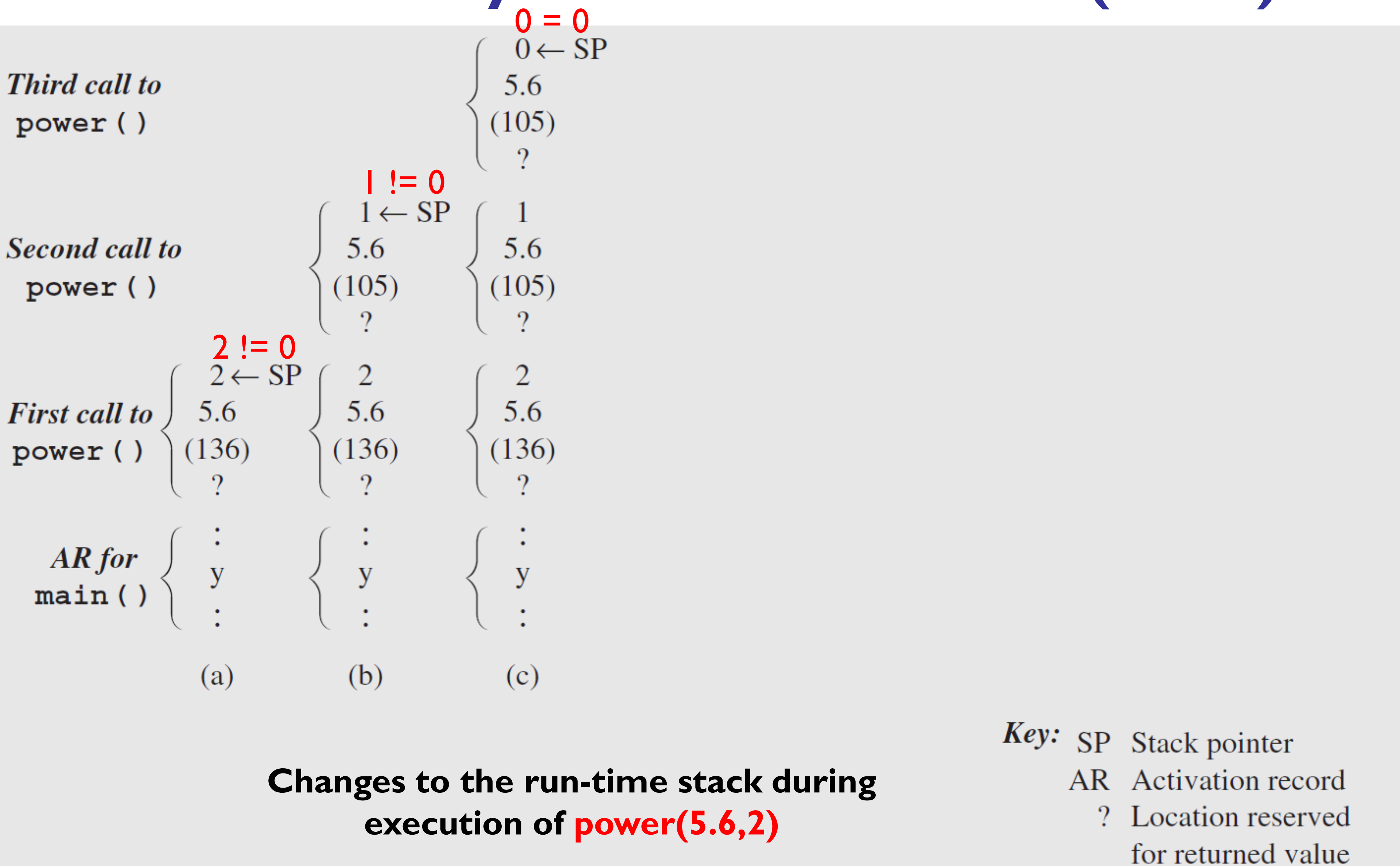
Anatomy of a Recursive Call (cont.)



Key: SP Stack pointer
AR Activation record
? Location reserved for returned value

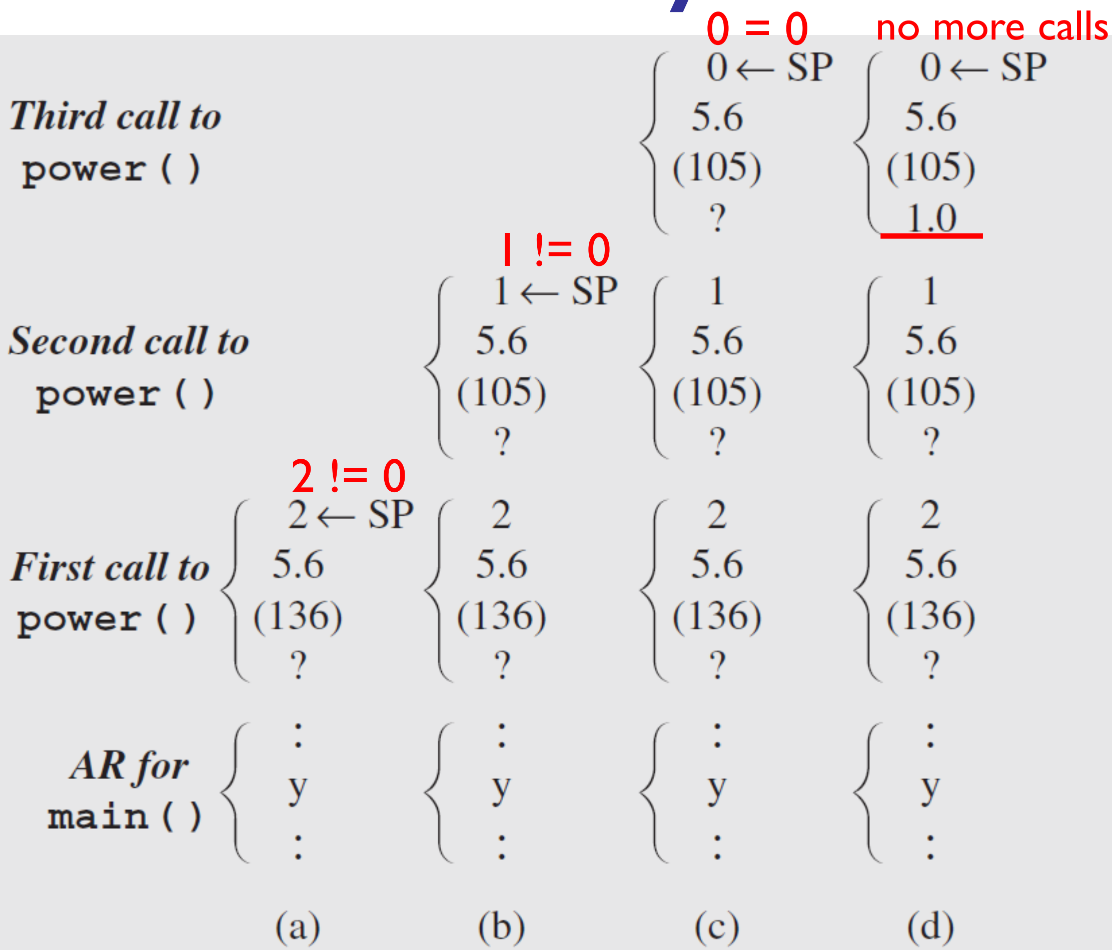
Changes to the run-time stack during execution of power(5.6,2)

Anatomy of a Recursive Call (cont.)



Changes to the run-time stack during execution of power(5.6,2)

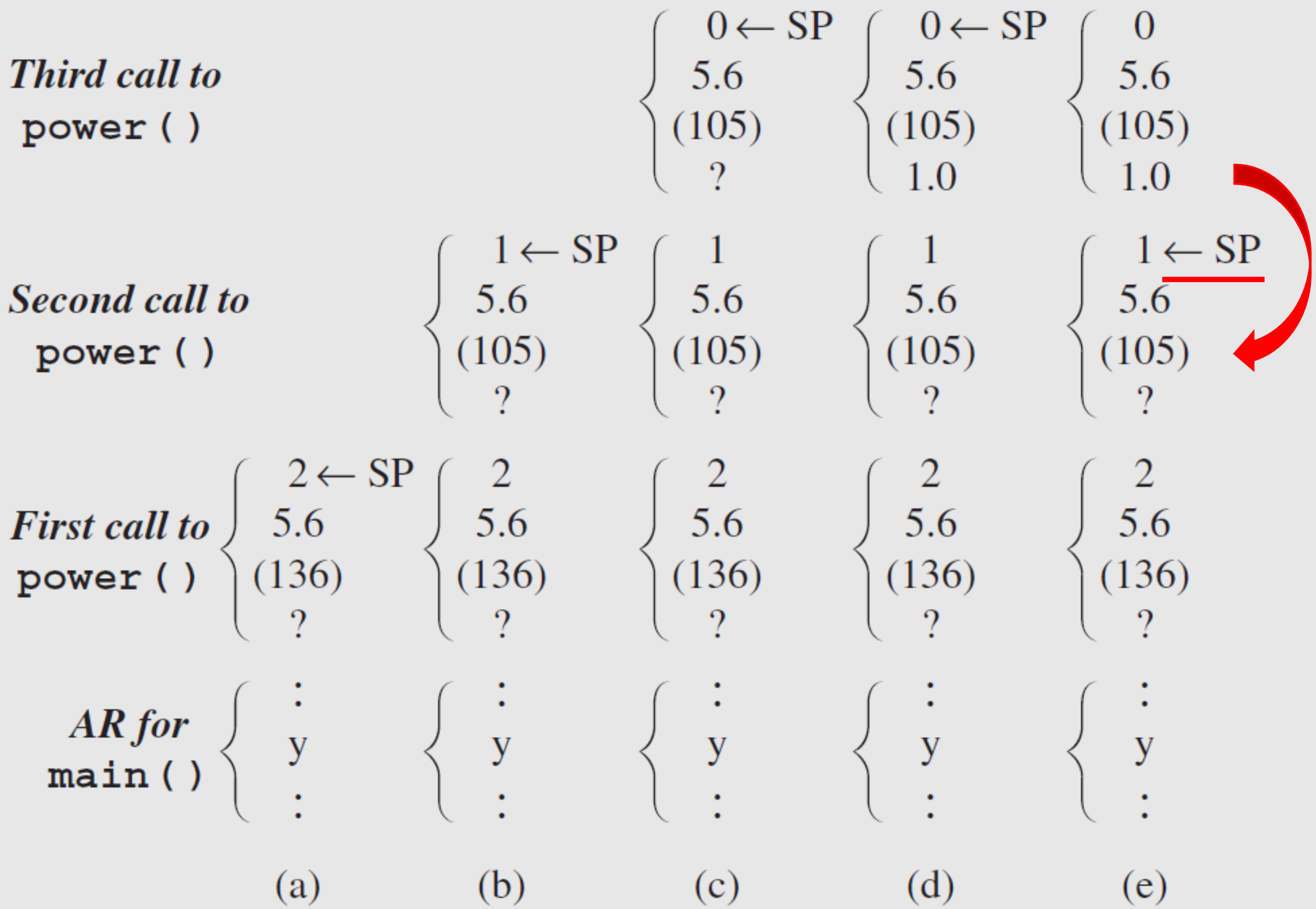
Anatomy of a Recursive Call (cont.)



Changes to the run-time stack during execution of power(5.6,2)

Key:
SP Stack pointer
AR Activation record
? Location reserved for returned value

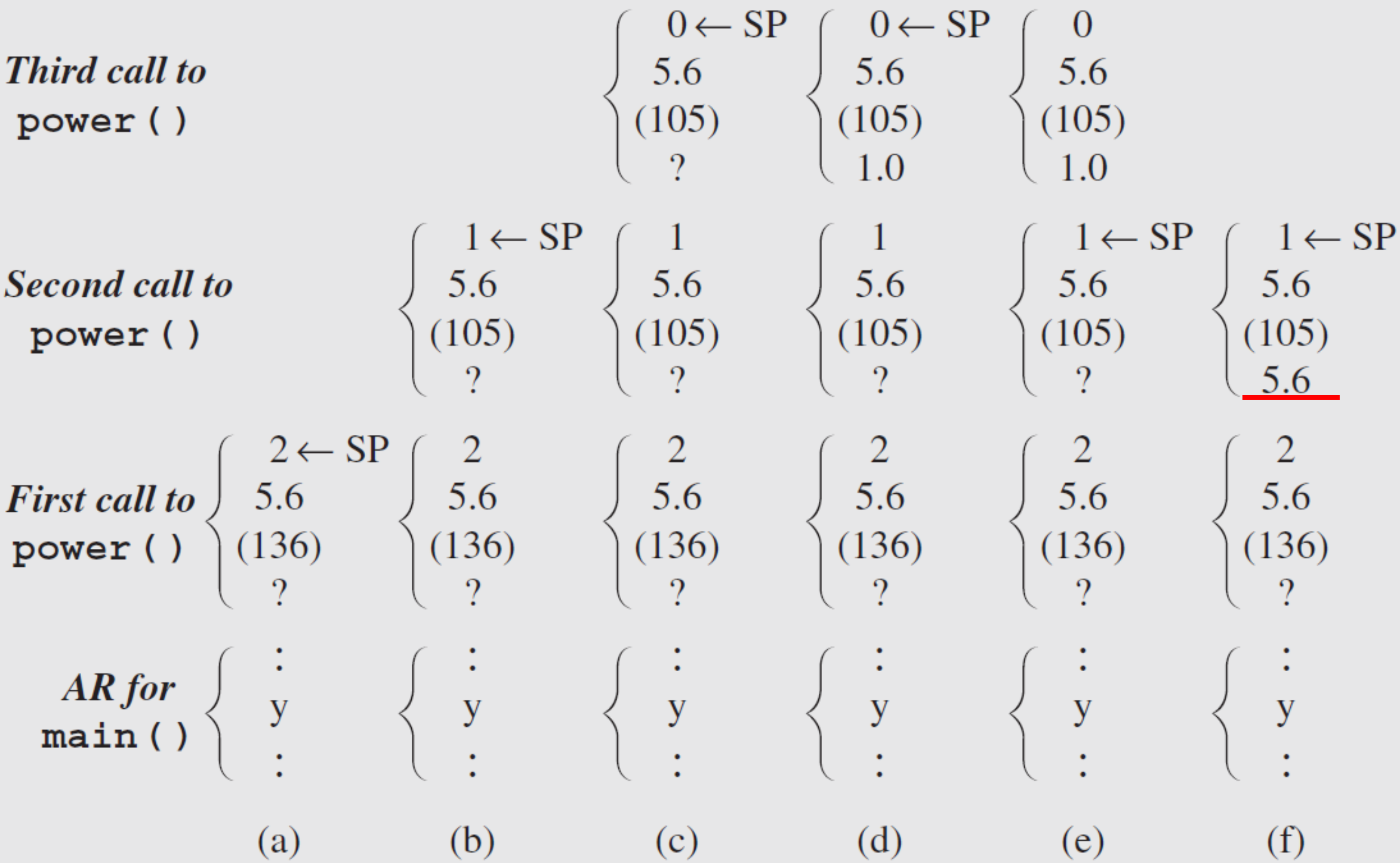
Anatomy of a Recursive Call (cont.)



Key: SP Stack pointer
AR Activation record
? Location reserved for returned value

Changes to the run-time stack during execution of power(5.6,2)

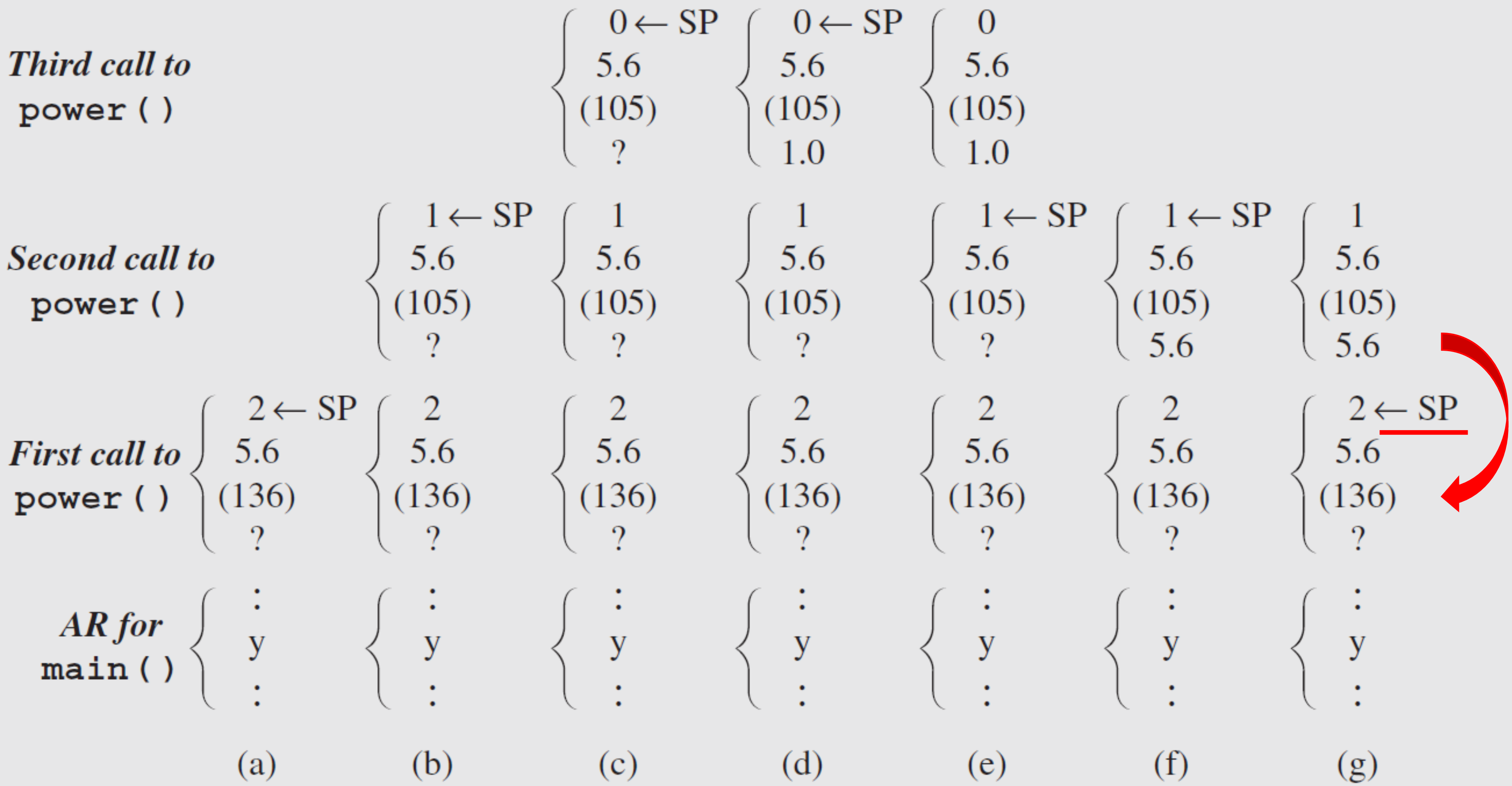
Anatomy of a Recursive Call (cont.)



Key: SP Stack pointer
AR Activation record
? Location reserved for returned value

Changes to the run-time stack during execution of power(5.6,2)

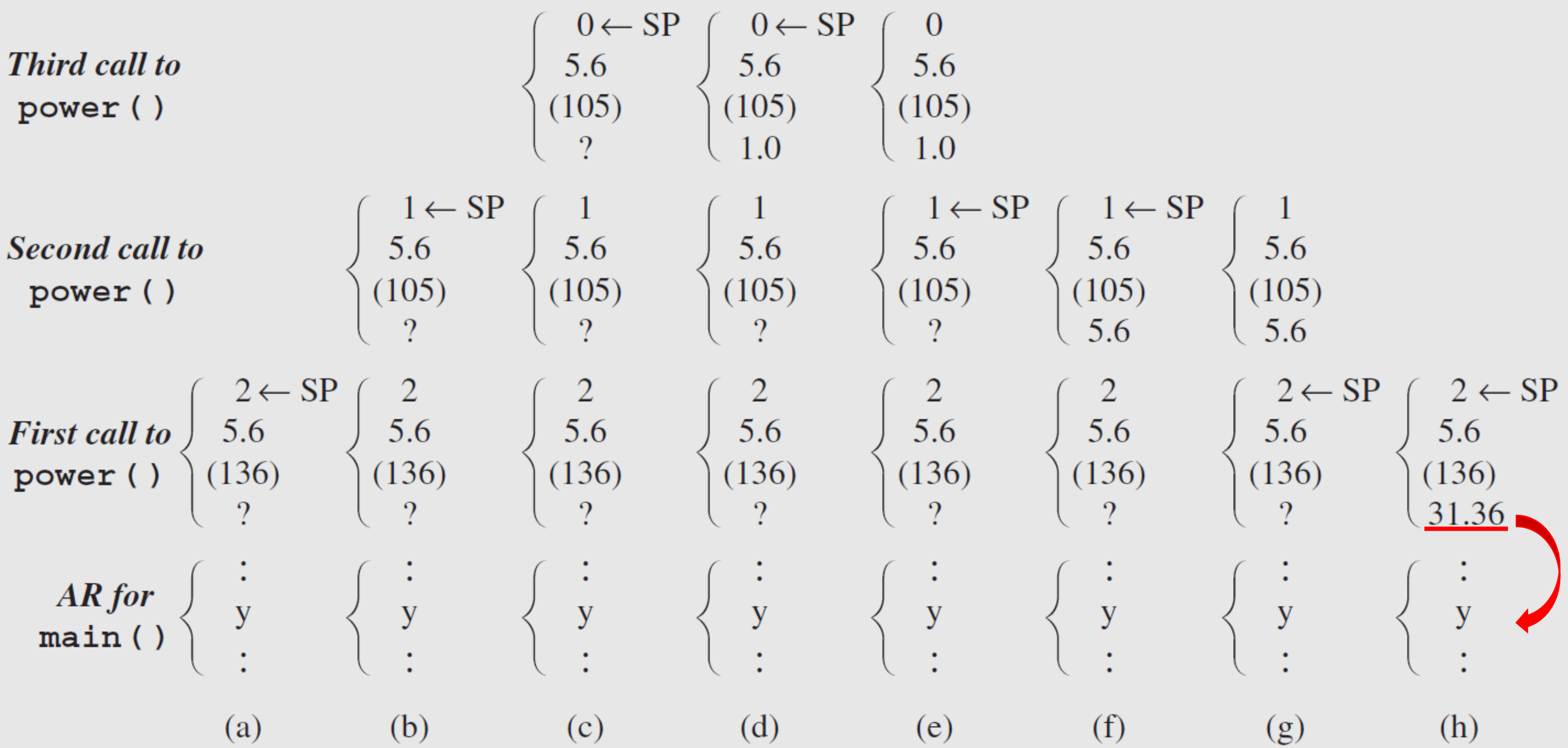
Anatomy of a Recursive Call (cont.)



Key: SP Stack pointer
AR Activation record
? Location reserved for returned value

Changes to the run-time stack during execution of power(5.6,2)

Anatomy of a Recursive Call (cont.)



Key:
SP Stack pointer
AR Activation record
? Location reserved for returned value

Changes to the run-time stack during execution of power(5.6,2)

Anatomy of a Recursive Call (cont.)

- Possible to implement the `power()` function in a ***non-recursive manner***??

```
double nonRecPower(double x, unsigned int n) {
   double result = 1;
   for (; n > 0; n--)
      result *= x;
   return result;
}
```

- comparing this to the recursive version,
  - the recursive code is more intuitive, closer to the specification, and simpler to code

Tail Recursion

- The nature of a recursive definition
  - the function contains a reference to itself
  - this reference can take on a number of different forms
- Starting with the simplest, ***tail recursion***
  - a **single** recursive call occurs **at the end of the function**
  - **no other statements** follow the recursive call
  - **no other recursive calls** prior to the call at the end of the function

Tail Recursion (cont.)

- e.g., a tail recursive function:

```
void tail(int i) {
   if (i > 0) {
     cout << i << '';
     tail(i-1);
   }
}
```

```
void nontail(int i) {
   if (i > 0) {
     nontail(i - 1);
     cout << i << '';
     nontail(i - 1);
   }
}
```

not tail recursion!

- Tail recursion,
  - a loop
  - can be replaced by an **iterative algorithm** to accomplish the same task

Tail Recursion (cont.)

- e.g., an iterative form of the function:

```
void iterativeEquivalentOfTail(int i) {
   for ( ; i > 0; i--)
      cout << i << '';
}
```

- any *advantage* in using tail recursion over iteration??

Nontail Recursion

- e.g., another type of recursion:

```
/* 200 */    void reverse() {
                char ch;
/* 201 */       cin.get(ch);
/* 202 */       if (ch != '\n') {
/* 203 */           reverse();
/* 204 */           cout.put(ch);
                }
             }
```

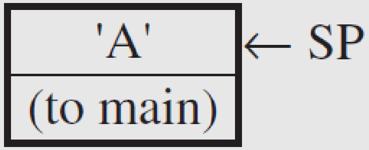
  - the recursive call precedes other code in the function
    - **nontail recursion**
  - display a line of input in **reverse order**
  - assuming the input, "ABC"

Nontail Recursion (cont.)

```
/* 200 */    void reverse() {
                char ch;
/* 201 */       cin.get(ch);
/* 202 */       if (ch != '\n') {
/* 203 */           reverse();
/* 204 */           cout.put(ch);
                }
             }
```

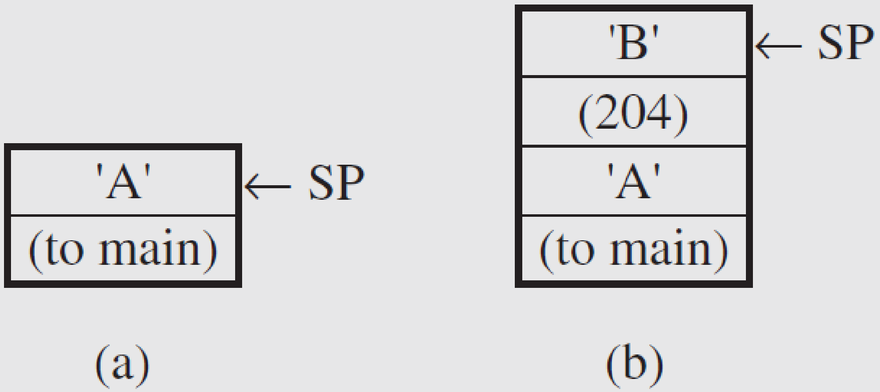
- The first time reverse() is called…
  - an **activation record** is created to store the **local variable** ch and the **return address** of the call in main()

Nontail Recursion (cont.)

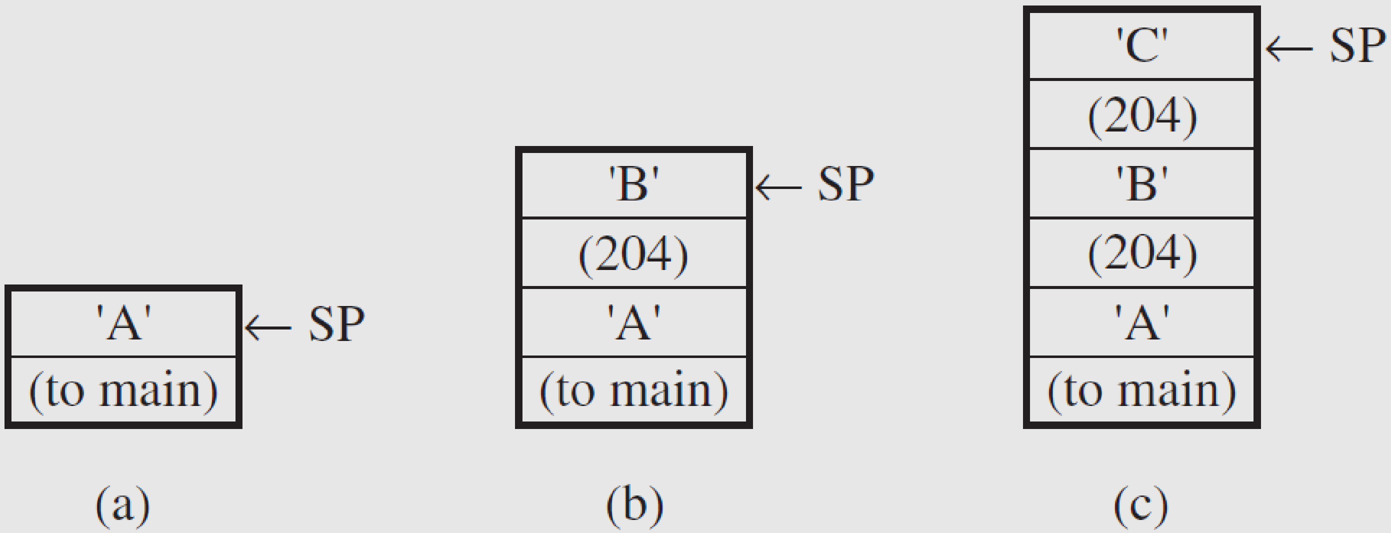


(a)

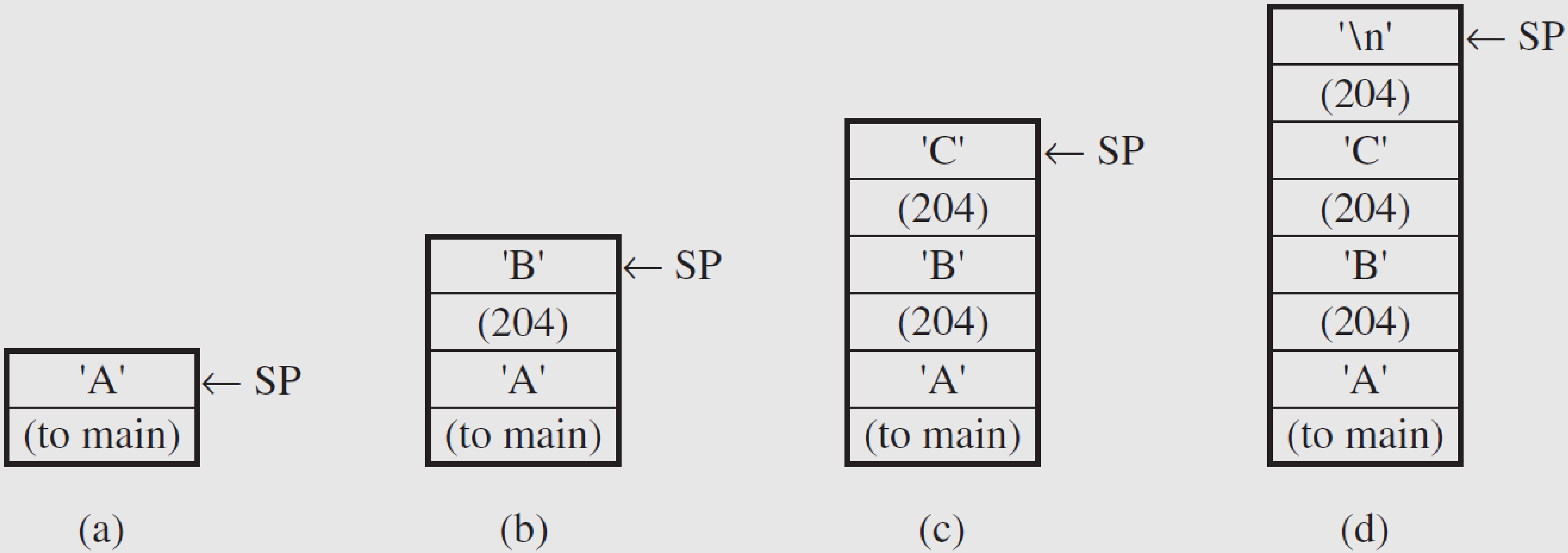
Nontail Recursion (cont.)



Nontail Recursion (cont.)



Nontail Recursion (cont.)



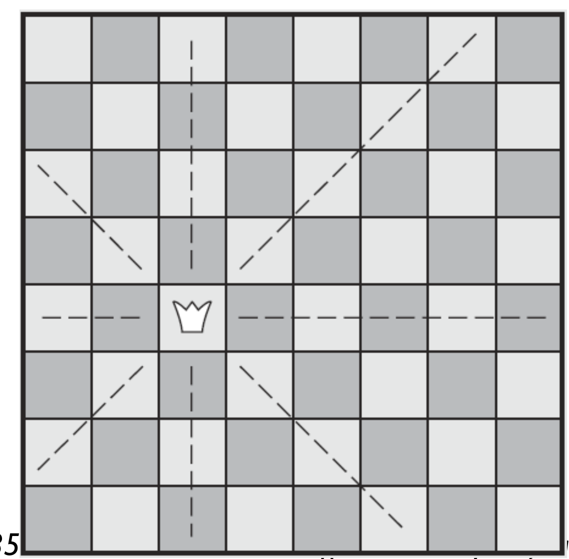
- When the end of line character is read,
  - the snapshot of stack appeared
  - **terminate** the current call
  - **popping** the last activation record off the **stack**
  - **resuming** the previous call

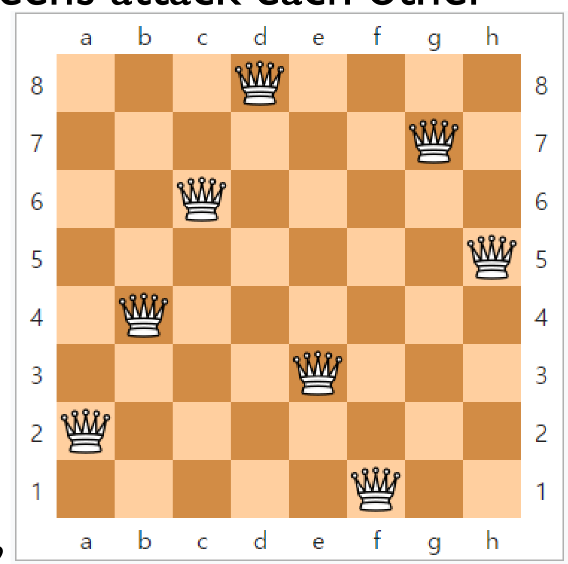
Backtracking

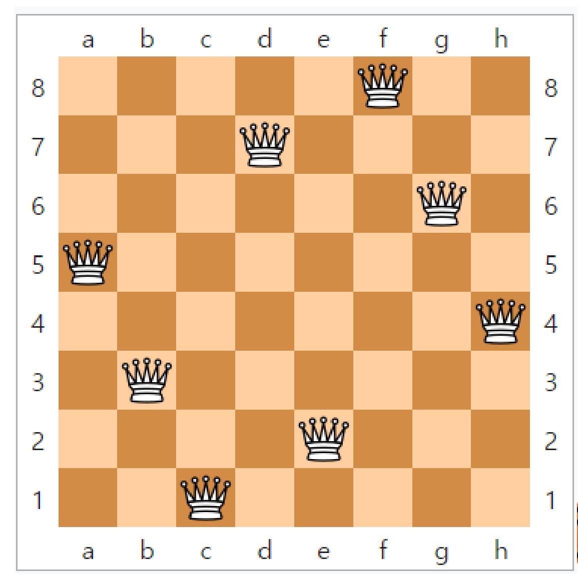
- In solving some problems
  - a situation arises where there are different ways leading from a given position
    - none of them known to lead to a solution
  - after trying one path unsuccessfully
    - *return* to the crossroads
    - *try* to find the solution using another path
  - ensure that a return is possible so that all paths can be tried
- ***Backtracking***
  - allows to systematically try all available paths from a certain point to solve the problem after some of paths lead to nowhere

Backtracking

- Potential applications of *backtracking*
  - artificial intelligence and optimization problems
  - e.g., ***The Eight Queens Problem*** – *no two queens share the same row, column, or diagonal*
    - try to place eight queens on a chessboard (8 x 8) in such a way
      - no two queens attack each other





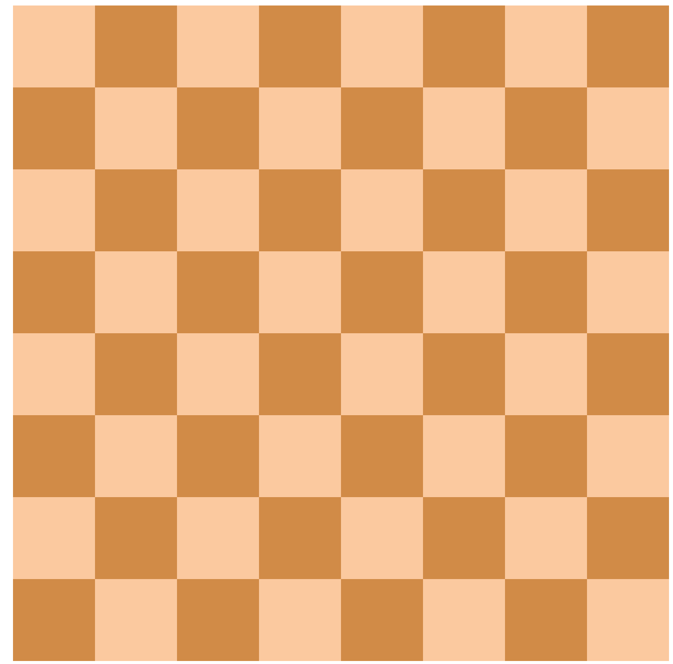


Backtracking (cont.)

- Place one queen at a time,
  - trying to make sure that the queens do not attack each other
- If at any point a queen cannot be successfully placed,
  - backtrack to the placement of the previous queen with different position
  - then, the next queen is tried again
- If no successful arrangement is found,
  - backtracks further
  - adjust the previous queen's predecessor, etc.

Backtracking (cont.)

```
putQueen(row)
    for every position col on the same row
        if position col is available
            place the next queen in position col;
            if (row < 8)
                putQueen(row+1);
            else success;
            remove the queen from position col;
```



- This algorithm will find all solutions, although some are symmetrical