Recursion

Data Structures – CS2001 – Fall 2021

Dr. Syed Ali Raza
School of Computer Science
National University of Computing & Emerging Sciences
Karachi Campus

Agenda

- Recursion Introduction
- Examples
- Internal Details
- Anatomy of Recursion (and Execution trace)
- Recursion Uses

Recursion

- In programming, a function calling itself is recursion.
- It is a way of solving a bigger (complex) problem by solving smaller problems of the same type.
- The idea is to decompose a problem into a smaller problem using some formula which can be further divided into more smaller problems using the same formula.
- But there should be a point at which it should be impossible to make the problem more smaller.

```
int func(){
    …
    func();
}
```

A Simple Recursion Program

Print n numbers:

```
void print(int n){
        if (n == 0)
                return;
        cout<<n;
        print(n-1);
}
int main() {
        int a=3;
        print(a);
        return 0;
}
```

Recursion

- A recursive function needs to have two parts.

Base case:

- The simplest case. You return a definite answer (value) here. A recursive call can’t be made here.
- Problem division into sub-problems ends here.

Recursive case:

- This is where you have a general formula to break problem into sub-problems. It makes (at least) one recursive call to itself. It comes closer to the base case at every step.

Recursion

- A recursion always goes through two phases:

A wind-up phase:

- When the base case is not satisfied, i.e. function calls itself.
- This phase carries on until we reach the base case.

An unwind phase:

- The recursively called functions return their values to previous "instances" of the function call
  - i.e. the last function returns to its parent (the 2nd last function), then the 2nd last function returns to the 3rd last function, and so on
- Eventually reaches the very first function, which computes the final value

The Musts of Recursion

- Your code must have a case for all valid inputs
- You must have a base case that makes no recursive calls
- When you make a recursive call it should be to a simpler instance and make forward progress towards the base case.
- The test for the base case must execute before the recursive function call.

A Simple Recursion Program

Sum from 1 to n:

Recursion based implementation:

```
int sum(int n){
        if (n == 1)
                return 1;
        return n + sum(n-1);
}
int main() {
        int a=3;
        cout<<sum(a);
        return 0;
}
```

Iteration based implementation:

```
int sum_i(int n){
        int sum=0;
        for (int i = 0; i < n; i++){
                sum += i;
        }
        return sum;
}
```

A Simple Recursion Program

Sum number between m and n:

Recursion based implementation:

```
int sum(int m, int n){
        if (m==n)
                return n;
        return m + sum2(m+1, n);
}
int main() {
        int m=2, n=4;
        cout<<sum2_i(m,n);
        return 0;
}
```

Iteration based implementation:

```
int sum2_i(int m, int n){
    int sum=0;
    for (int i = m; i <= n;
i++){
        sum += i;
        }
        return sum;
}
```

Student Count in a Column in Your Class

```
int numStudentsBehind(Student curr) {
     if (noOneBehind(curr)) {
          return 0; }
     else {
          Student personBehind = curr.getBehind();
          return numStudentsBehind(personBehind) + 1;
          }
}
```

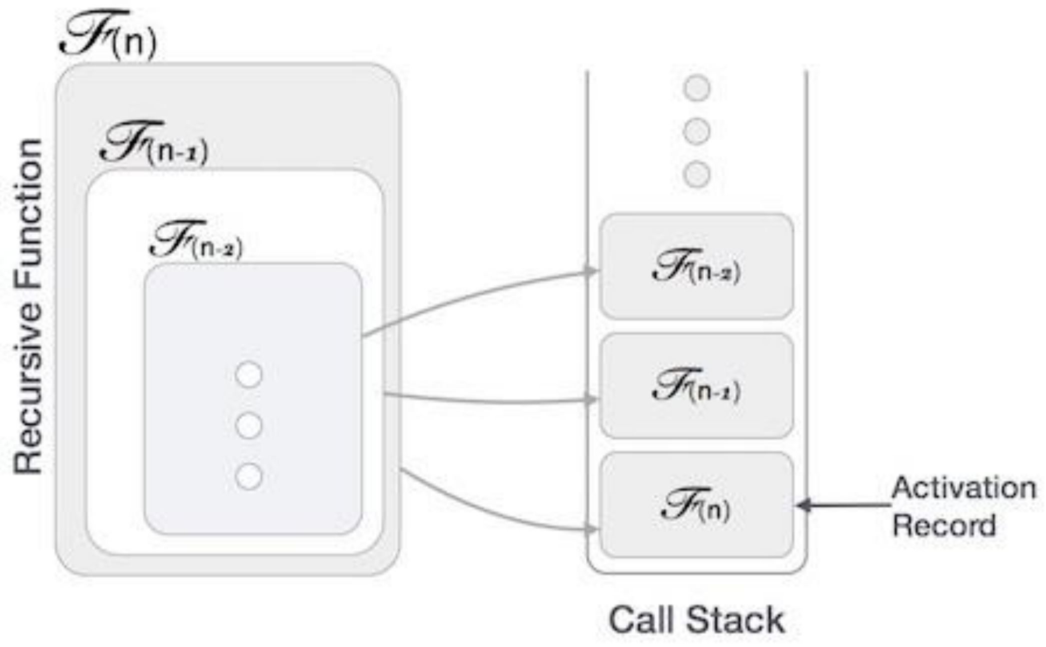
Recursion – Internal Working

- Whenever a function (**caller**) calls another function (**callee**) or itself as callee, the caller function transfers execution control to the callee.
- The caller function has to suspend its execution temporarily and resume later when the execution control returns from the callee function.
- Here, the caller function needs to start exactly from the point of execution where it puts itself on hold.
- It also needs the exact same data values it was working on.
- For this purpose, an activation record (or stack frame) is created for the caller function.

Recursion – Internal Working

Stack Frame or Activation record

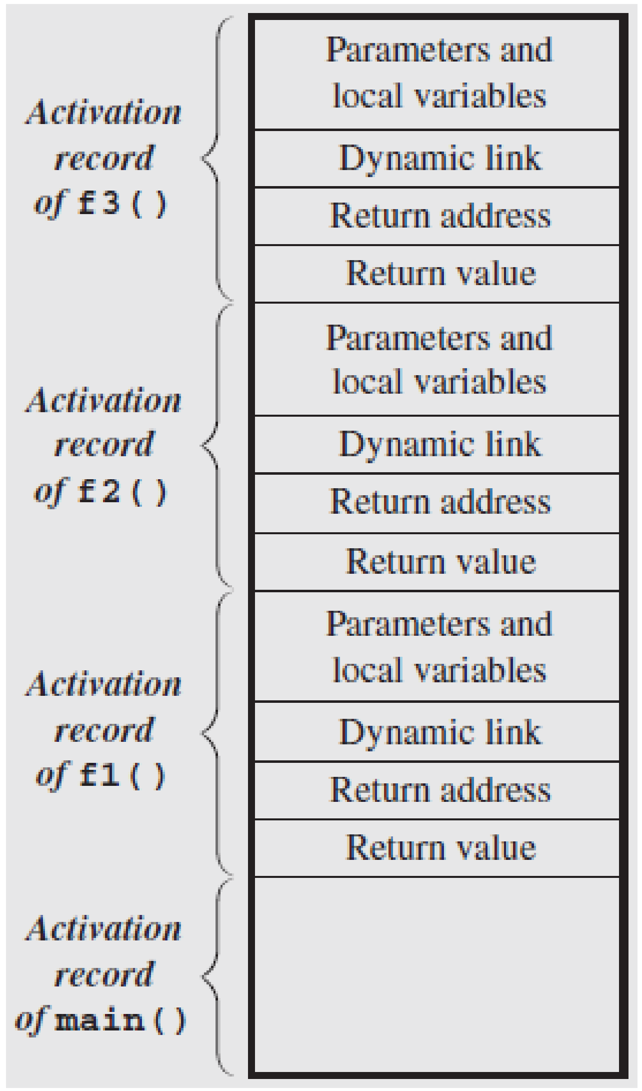
- When a function is called its activation record is stored in the stack.
- It holds the current state of the function.
- It is dynamically allocated at function entry and deallocated upon exiting.
- It contains information private to the function.



Recursion – Internal Working

An activation record contains,

- Values for all parameters to the function, location of the first cell if an array is passed or a variable is passed by reference, and copies of all other data items.
- Local variables that can be stored elsewhere, in which case, the activation record contains only their descriptors and pointers to the locations where they are stored.
- The return address to resume control by the caller, the address of the caller's instruction immediately following the call.
- A dynamic link, which is a pointer to the caller's activation record.
- The returned value for a function not declared as void. Because the size of the activation record may vary from one call to another, the returned value is placed right above the activation record of the caller.


Activation record of f3()
Parameters and local variables
Dynamic link
Return address
Return value
Activation record of f2()
Parameters and local variables
Dynamic link
Return address
Return value
Activation record of f1()
Parameters and local variables
Dynamic link
Return address
Return value
Activation record of main()

Anatomy of Recursion

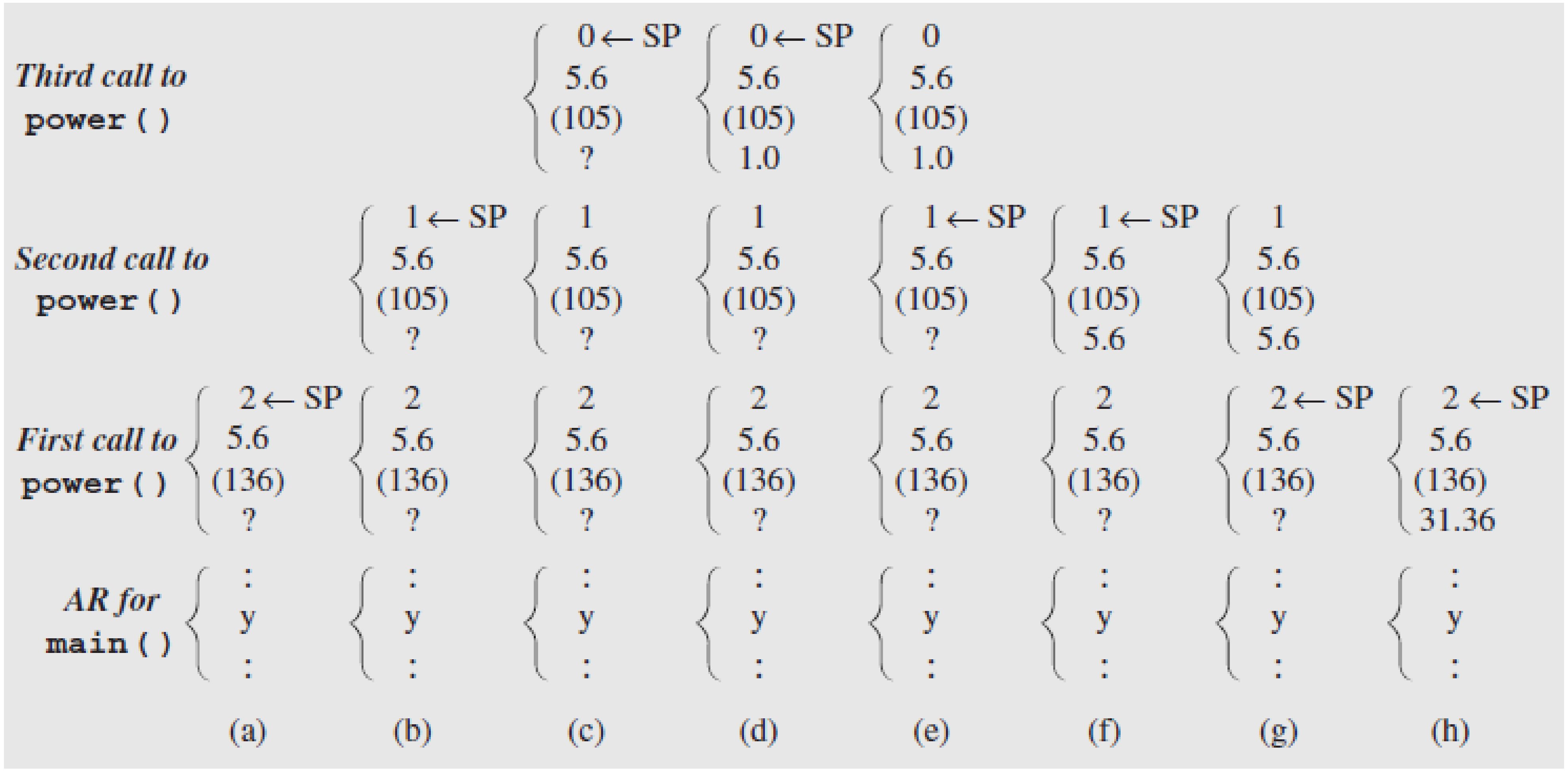
- Recursion based power function (raising any number *x* to a nonnegative integer power *n*) is defined as,

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

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

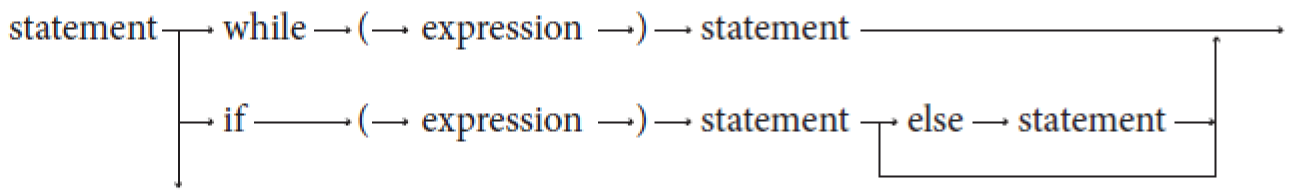
```c
           int main()
           { ...
/* 136 */   y = power(5.6,2);
              ...
           }
```

Anatomy of Recursion



Recursion Uses

- The recursive version increases program readability, improves self-documentation, and simplifies coding.
- Many problems naturally appear recursive and a recursion based solution is their best solution (like , Quicksort, Tower of Hanoi, etc.)
- It is very natural mechanism for processing recursive data structures (like for tree traversals.)
- Recursive definitions are frequently used to define sequences of numbers. Like, factorial sequences. 0,1,2,3,4,.. to 1,1,2,6,24,...
- Recursive definitions are used extensively in the specification of the grammars of programming languages.



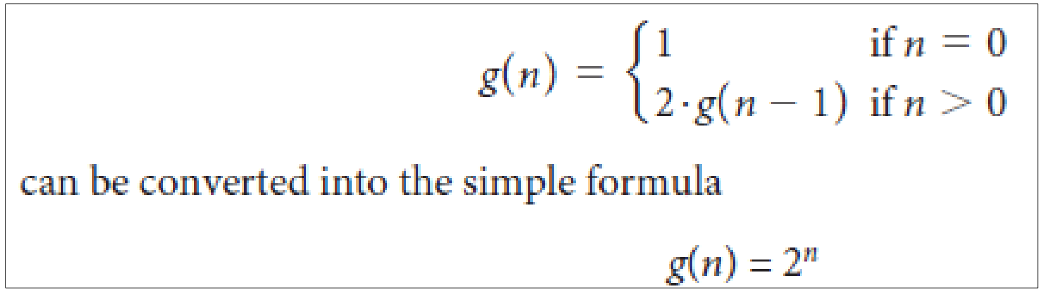
Recursion Downsides

- Many (simple), but not all, recursions essentially accomplish a loop (iterations)
- In general, an iterative version of a method will execute more efficiently in terms of time and space than a recursive version.
- This is because the overhead involved in entering and exiting a function in terms of stack I/O is avoided in iterative version.
- Also, recursion can increase memory usage and can cause stack overflow.
- The form used for sub-problem formation in recursion can be difficult to understand.

Recursion Alternatives

Common practices,

- If we convert our recursion to iterative version, we will generally do so.
- Find an equivalent definition or formula that makes no references to other elements of the sequence. Generally, finding such a formula is a difficult problem that cannot always be solved. But the formula is preferable to a recursive definition.



$$g(n) = \begin{cases} 1 & \text{if } n = 0 \\ 2 \cdot g(n-1) & \text{if } n > 0 \end{cases}$$

can be converted into the simple formula

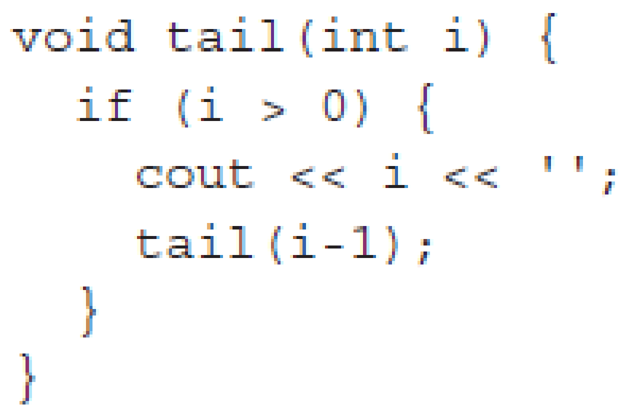
$$g(n) = 2^n$$

Types of Recursion

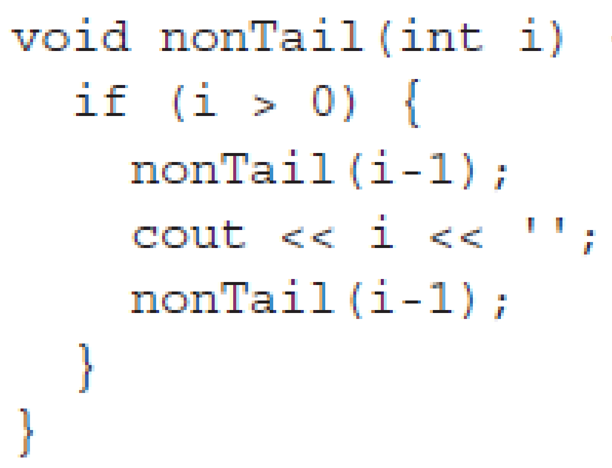
- Tail vs. Non Tail Recursion
- Direct vs. InDirect Recursion
- Nested Recursion
- Excessive Recursion

Tail Recursion

- When the call is made, there are no statements left to be executed by the function
- The recursive call is not only the last statement but there are no earlier recursive calls, direct or indirect.



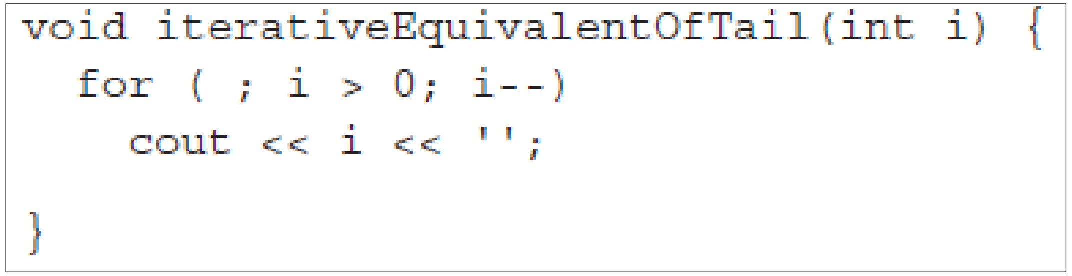
```
void tail(int i) {
  if (i > 0) {
    cout << i << '';
    tail(i-1);
  }
}
```



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

Tail Recursion

- A tail recursion can be easily replaced by a loop.
- In languages where there is no loop construct, tail recursion can be used.



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

}
```

```
void f1(int n) {
   cout<< n << "  ";
   if (n > 0)
      f1(n - 1);
}
```

```
void f1(int n) {
   for( int k = n; k >= 0; k--)
        cout<< k << "  ";
}
```

Non-Tail Recursion

- A recursive call is not the last statement in the recursive function.
- Some part of the function is evaluate after the return from the calle.

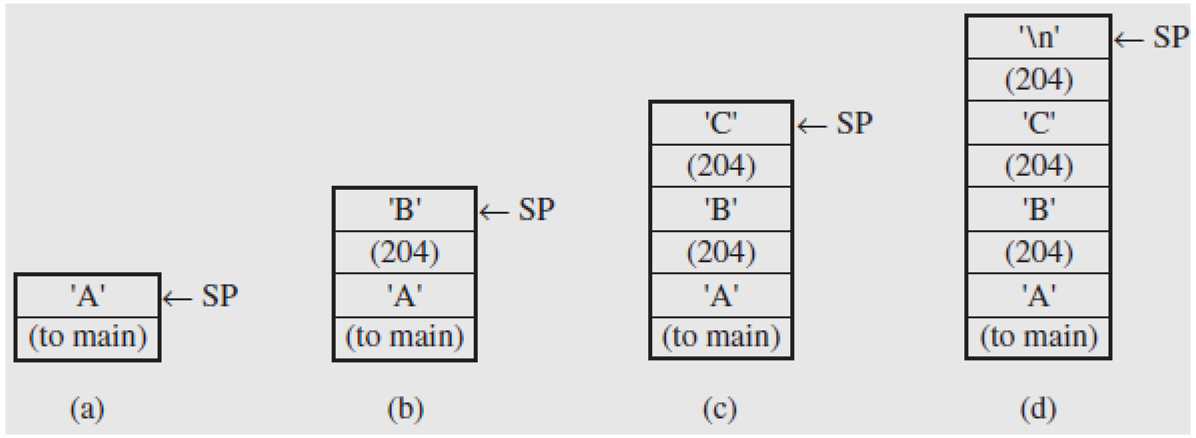
```
void print(int n){
	if (n == 0)
		return;
	print(n-1);
	cout << n << " ";
}
int main() {
	print(3);
	return 0;
}
```

```
int func(int n){
	if (n == 1)
		return 0;
	return 1 + func(n/2);
}
int main() {
	cout<< func(8);
	return 0;
}
```

Non-Tail Recursion

- Printing an input line in reverse order.

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



Non-Tail Recursion

- The transformation of nontail recursion into iteration usually involves the explicit handling of a stack but it can be done using some loops.

```
void f3(int n) {
  if (n > 6) {
    cout<< 2*n << " ";
      f3(n - 2);
  } else if (n > 0) {
    cout<< n << "  ";
    f3 (n - 1);
  }
}
```

$\longrightarrow$

```
void f3 (int n) {
  while (n > 0) {
    if (n > 6) {
       cout<< 2*n << "  ";
       n = n - 2;
    } else if (n > 0) {
       cout<< n << "  ";
        n = n - 1;
    }
  }
}
```

Indirect Recursion

- A function is called direct recursive if it directly calls itself again.
- In indirect recursion, a function (say f1) calls another function (say f2), then this function calls the first function (f1) (they will be called *mutually recursive*).
- f1 can call itself indirectly via a chain of other calls. The chain of intermediate calls can be of an arbitrary length, as in:
  - f() -> f1() -> f2() -> ... -> fn() -> f()
- f1 can call itself indirectly through different chains.
  - f() -> g1() -> g2() -> ... -> gm() -> f()

Indirect Recursion

Make odd even and even odd

```
void even();
void odd();
int n=1;

void odd() {
        if (n <= 10){
                cout << n + 1;
                n++;
                even();
        }
        return;
}
```

```
void even() {
   if (n<=10){
       cout<< n-1;
       n++;
       odd();
   }
   return;
}
int main() {
       odd();
       return 0;
}
```

Indirect Recursion

- Example concerning formulas calculating the trigonometric functions sine, cosine, and tangent:

$$\sin(x) = \sin\left(\frac{x}{3}\right)\cdot\frac{(3 - \tan^2(\frac{x}{3}))}{(1 + \tan^2(\frac{x}{3}))}$$

$$\tan(x) = \frac{\sin(x)}{\cos(x)}$$

$$\cos(x) = 1 - \sin\left(\frac{x}{2}\right)$$

$$\sin(x) \approx x - \frac{x^3}{6}$$ small values of *x*

```
double sin(double x){
   if(x < 0.0000001)
      return x - (x*x*x)/6;
   else{
      double y = tan(x/3);
      return sin(x/3)*((3 - y*y)/(1 + y*y));
   }
}
double tan(double x){
   return sin(x)/cos(x);
}
double cos(double x){
   double y = sin(x);
   return sqrt(1 - y*y);
}

int main() {
       cout<< sin(1);
       return 0;
}
```

Nested Recursion

- Nested recursion occurs when a method is not only defined in terms of itself; but it is also used as one of the parameters:
- Example: The Ackerman function (It grows faster than a multiple exponential function.)

$$A(n, m) = \begin{cases} m + 1 & \text{if } n = 0 \\ A(n - 1, 1) & \text{if } n > 0, m = 0 \\ A(n - 1, A(n, m - 1)) & \text{otherwise} \end{cases}$$

```
long Ackmn(long n, long m){
    if (n == 0)
         return m + 1;
    else if (n > 0 && m == 0)
          return Ackmn(n - 1, 1);
     else
         return Ackmn(n - 1, Ackmn(n, m - 1));
}
```

Excessive Recursion

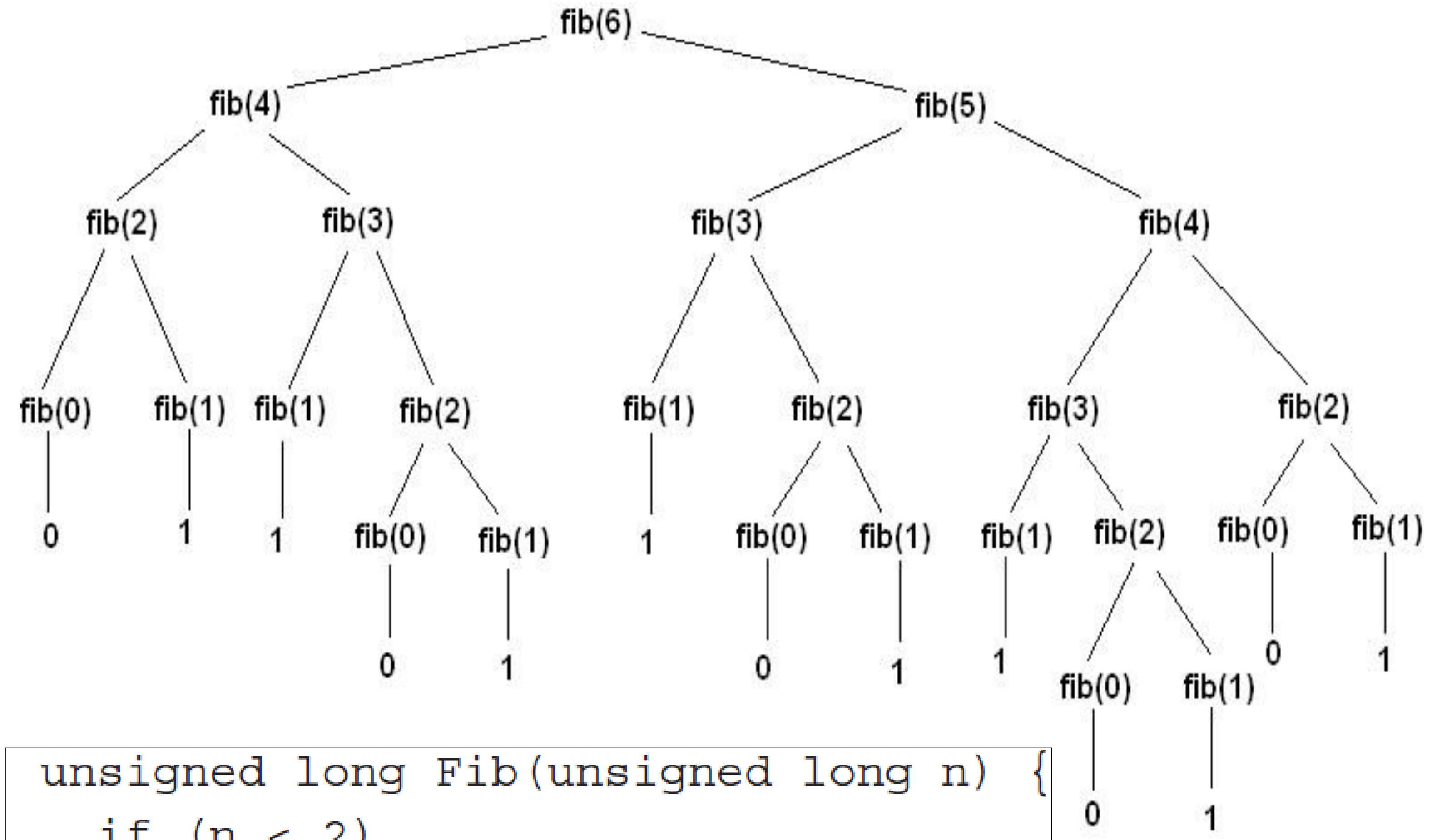
- A recursive method is excessively recursive if it repeats computations for some parameter values.
- Consider Fibonacci numbers where first two numbers are 0 and 1 then any number in the sequence is the sum of its two predecessors.
- Example: The call fib(6) results in two repetitions of f(4). This in turn results in repetitions of fib(3), fib(2), fib(1) and fib(0):

$$\mathrm{Fib}(n) = \begin{cases} n & \text{if } n < 2 \\ \mathrm{Fib}(n-2) + \mathrm{Fib}(n-1) & \text{otherwise} \end{cases}$$

The sequence produced by the definition is
0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, . . .

Excessive Recursion

- Tree of calls for Fibonacci number.
- To find fib(6)=8, fib() is called 25 times to determine the seventh element
- Recursive based program is extremely inefficient
- fib(26) takes quarter of a million calls and fib(31) takes nearly 3 million calls.



```
unsigned long Fib(unsigned long n) {
   if (n < 2)
     return n;
// else
     return Fib(n-2) + Fib(n-1);
 }
```

Solve Using Recursion

- **Problem: Path finder**
- Find the path from start to the end. You are given a 2d array representing a maze. Following is an example of a 4x4 maze,

```
0, 1, 1, 1
0, 1, 1, 1
0, 0, 1, 1
1, 0, 0, 2
```

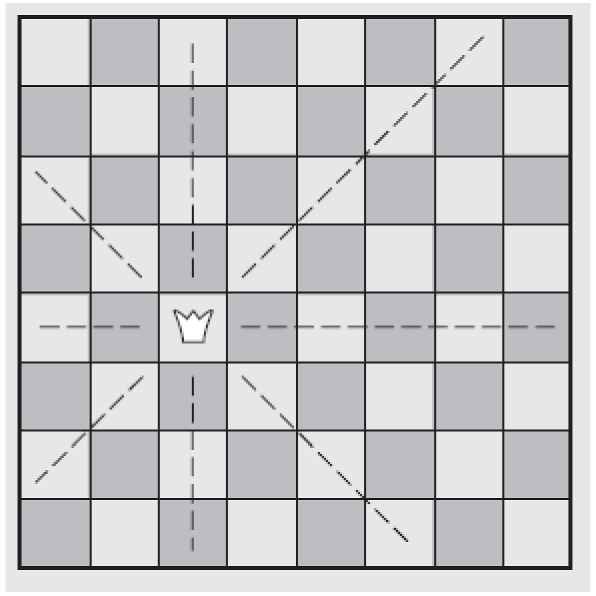
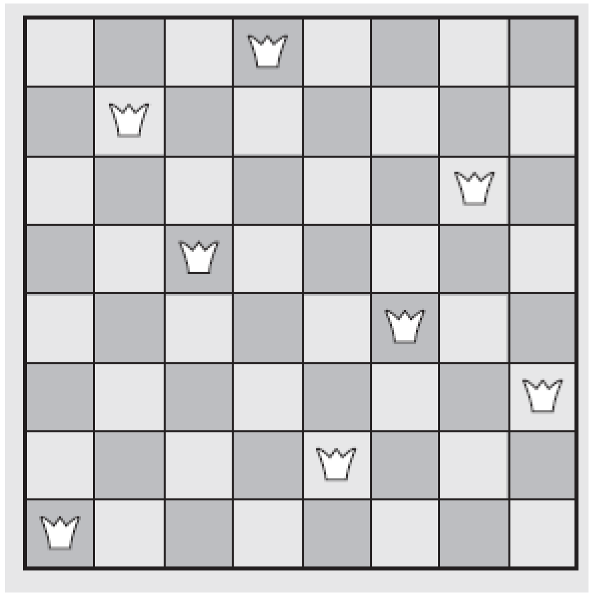
- Start top left, goal bottom right.
- Only down and right steps are allowed.
- Note, 0 shows an empty space, 1 shows an obstacle, 2 shows the goal. There exists only one path from start to end.

Backtracking

- Sometimes when solving a problem we are at crossroads. There can be multiple paths we can take and we have to decide which path we should follow to find the solution.
- However, the path which we select may end-up in a dead-end (no possible solution).
- In such a scenario, we want to go back to the point from where took the decision (which has failed) and try a different path.
- Backtracking is a method which allows to go back and try a different path (among the possible paths).
- It allows us to systematically try all available avenues from a certain point after some of them lead to nowhere.

Eight Queens Problem

- Place eight queens on a chessboard in such a way that no queen is attacking any other.
- A queen can take another piece if it lies on the same row, on the same column, or on the same diagonal as the queen.
- We try to put the first queen on the board, then the second so that it cannot take the first, then the third so that it is not in conflict with the two already placed, and so on.

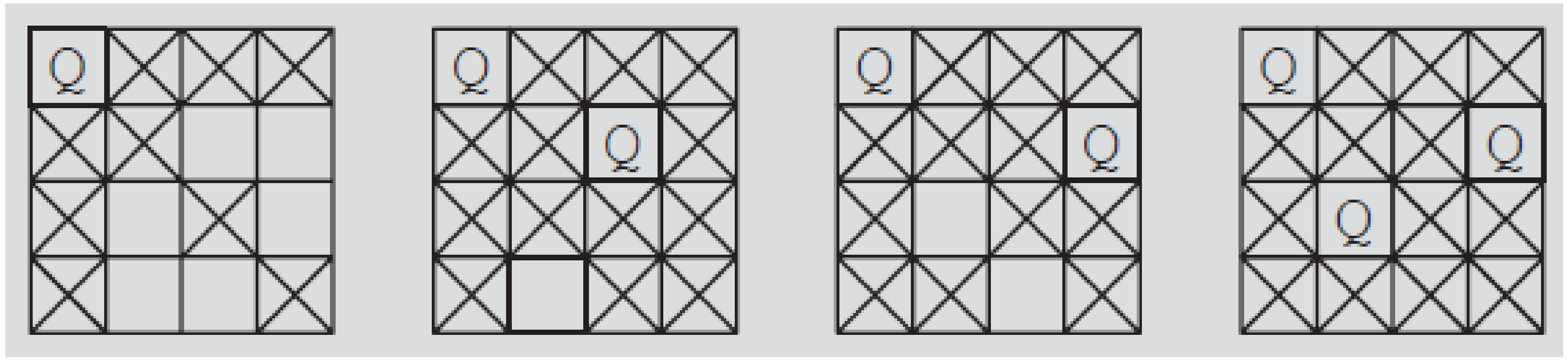
Eight Queens Problem

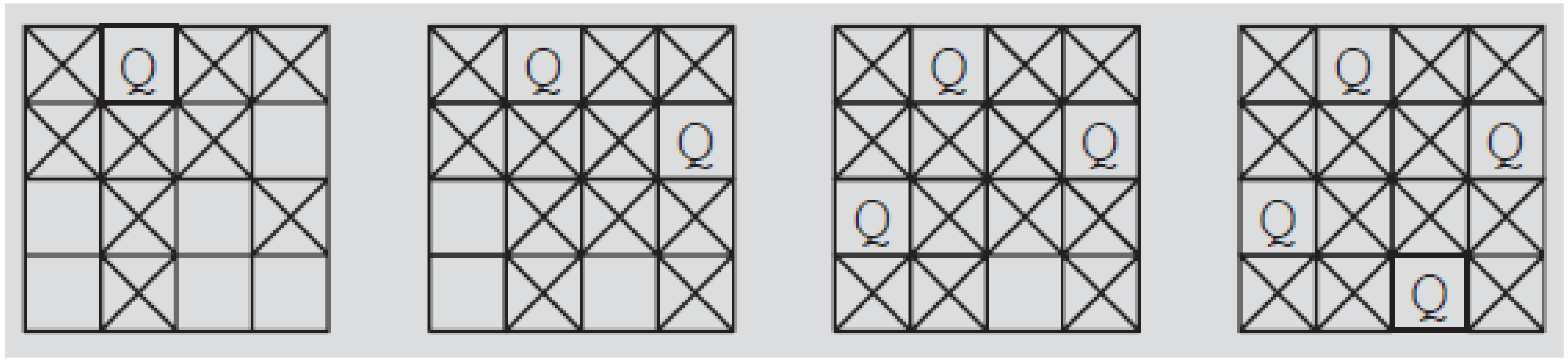
- What happens if, for instance, the sixth queen cannot be placed in a nonconflicting position?
- We backtrack and try some untried avenues.
- Recursion provides a natural implementation of backtracking.

```
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;
```

Eight Queens Problem

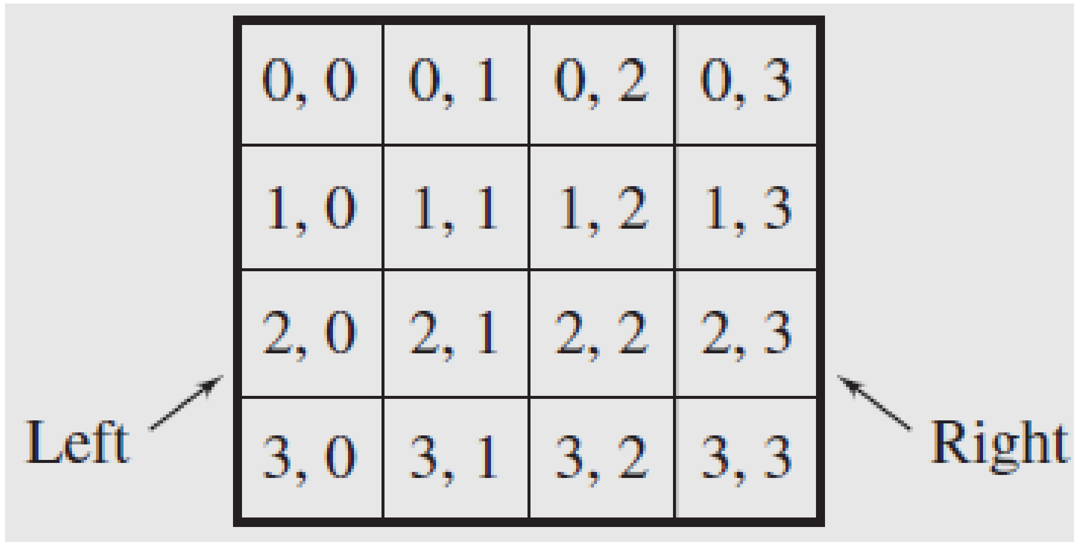
- First consider a 4 × 4 chessboard. Later, make changes to accommodate a regular board.





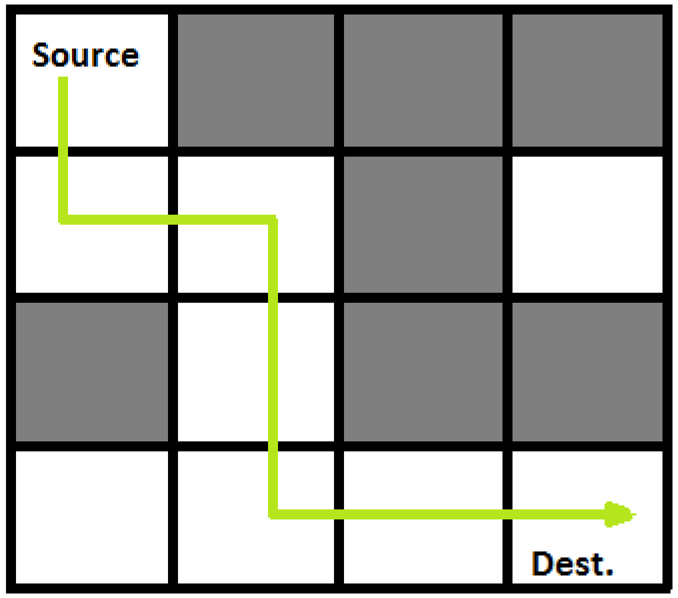
Eight Queens Problem

- First consider a 4 × 4 chessboard. Later, make changes to accommodate a regular board. (Book’s solution idea)



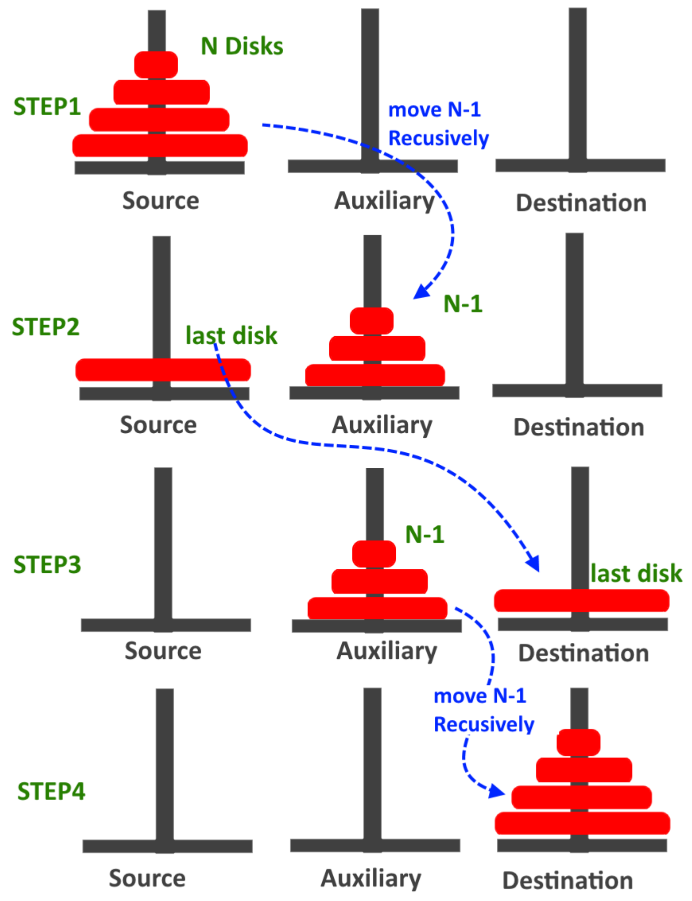
Rat in a Maze

- A Maze is given as N*N binary matrix of blocks where source block is the upper left most block i.e., maze[0][0] and destination block is lower rightmost block i.e., maze[N-1][N-1]. A rat starts from source and has to reach the destination. The rat can move only in two directions: forward and down.
- In the maze matrix, 0 means the block is a dead end and 1 means the block can be used in the path from source to destination.



Tower of Hanoi

- We have three rods and n disks. The objective is to move the entire stack to the destination rod from the source rod using the auxiliary rod. Rules:
    - Only one disk can be moved at a time.
    - Each move consists of taking the upper disk from one of the stacks and placing it on top of another stack i.e. a disk can only be moved if it is the uppermost disk on a stack.
    - No disk may be placed on top of a smaller disk.



N Disks
STEP1
move N-1
Recusively
Source
Auxiliary
Destination
STEP2
N-1
last disk
Source
Auxiliary
Destination
STEP3
N-1
last disk
Source
Auxiliary
Destination
move N-1
Recusively
STEP4
Source
Auxiliary
Destination