CSC230

Intro to C++ Lecture 10

Outline

- Recursion
- Lab 6 / Project 2 discussion

Segfault

- Segmentation Faults usually related to miss-use on memory
  - GDB
    - set breakpoint
    - run the program line by line
  - Valgrind
  - g++ -g .cpp
  - examples

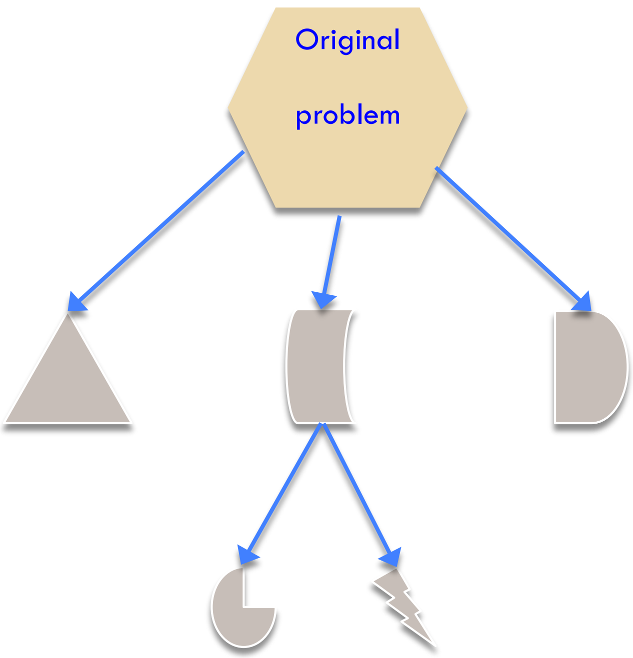
Outline

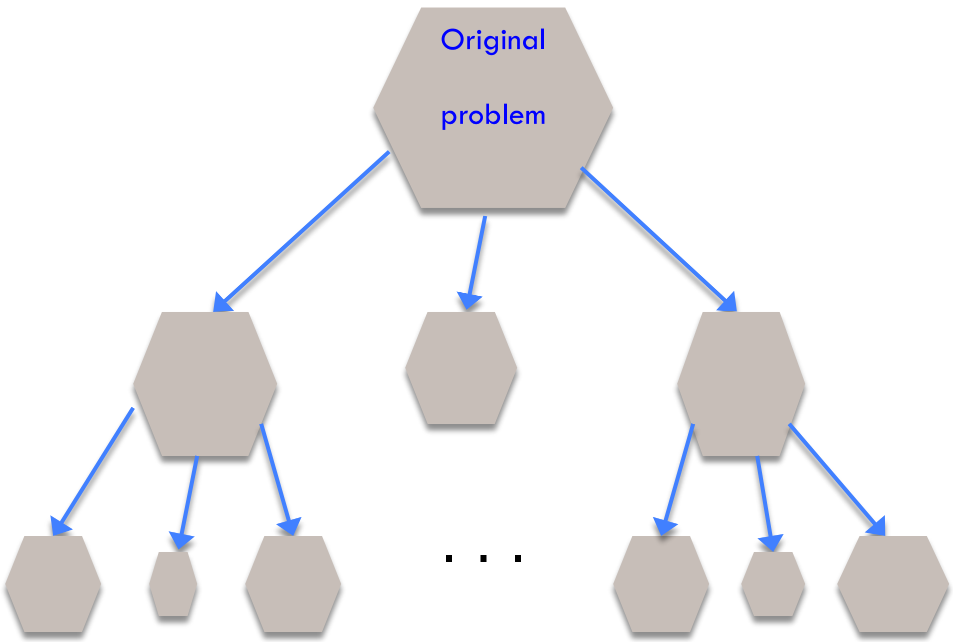
- Recursion

- Lab 6 / Project 2 discussion

How to solve a problem?

Divide and conquer





The focus of this lecture

Recursion

Arises in two forms in computer science

- Recursion as a *mathematical* tool for defining a function in terms of itself in a simpler case

- Recursion as a *programming* tool

Mathematical **induction** is used to prove that a recursive function works correctly. This requires a good, precise function specification.

See this in a later lecture.

Example, Sum the digits in a number

1	8	7	5	2	6

1 + 8 + 7 + 5 + 2 + 6 = 29

What is the **simple case** ? (**base case**)

4

How can we **reduce** the complexity of the problem? (**reduction step**)

1	8	7	5	2	6

1	8	7	5	2	+	6

Example: Sum the digits in a number

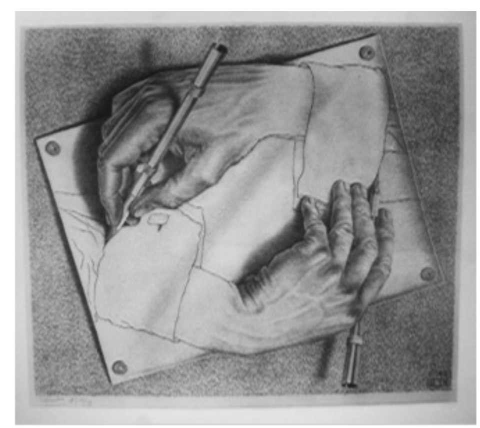
How to do it?

```
#include <iostream>
using namespace std;

/* return sum of digits in n.
 * Precondition:  n >= 0 */
static int sum(int n){
   if (n < 10) return n;
      // { n has at least two digits }
      // return first digit + sum of rest
      return n%10  +  sum(n/10);
}

int main() {
   cout << sum(187526) << endl;
}
```

sum calls itself!



50

- E.g. sum(187526) = 29

Palindrome

Palindrome: a sequence of characters which reads the same **backward** or **forward.**

racecar ✔

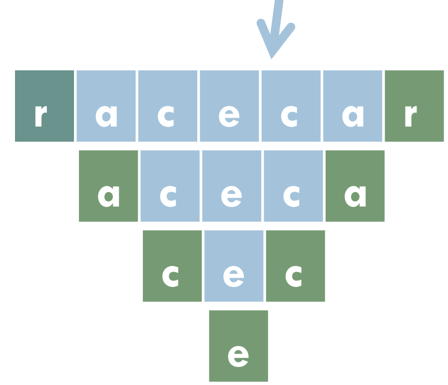
tcnj ✗

Q: How to decide whether a string is palindrome?
A: A palindrome is symmetric.

Q: How to determine whether a string is symmetric?
A: ….

Q: How to determine whether a string is symmetric?
A: If (the first char and the last char are euqual) && (the substring in the middle is symmetric)

Substring from s[1] to s[n-1]



Example: Is a string a palindrome?

How to check if a string is a palindrome?

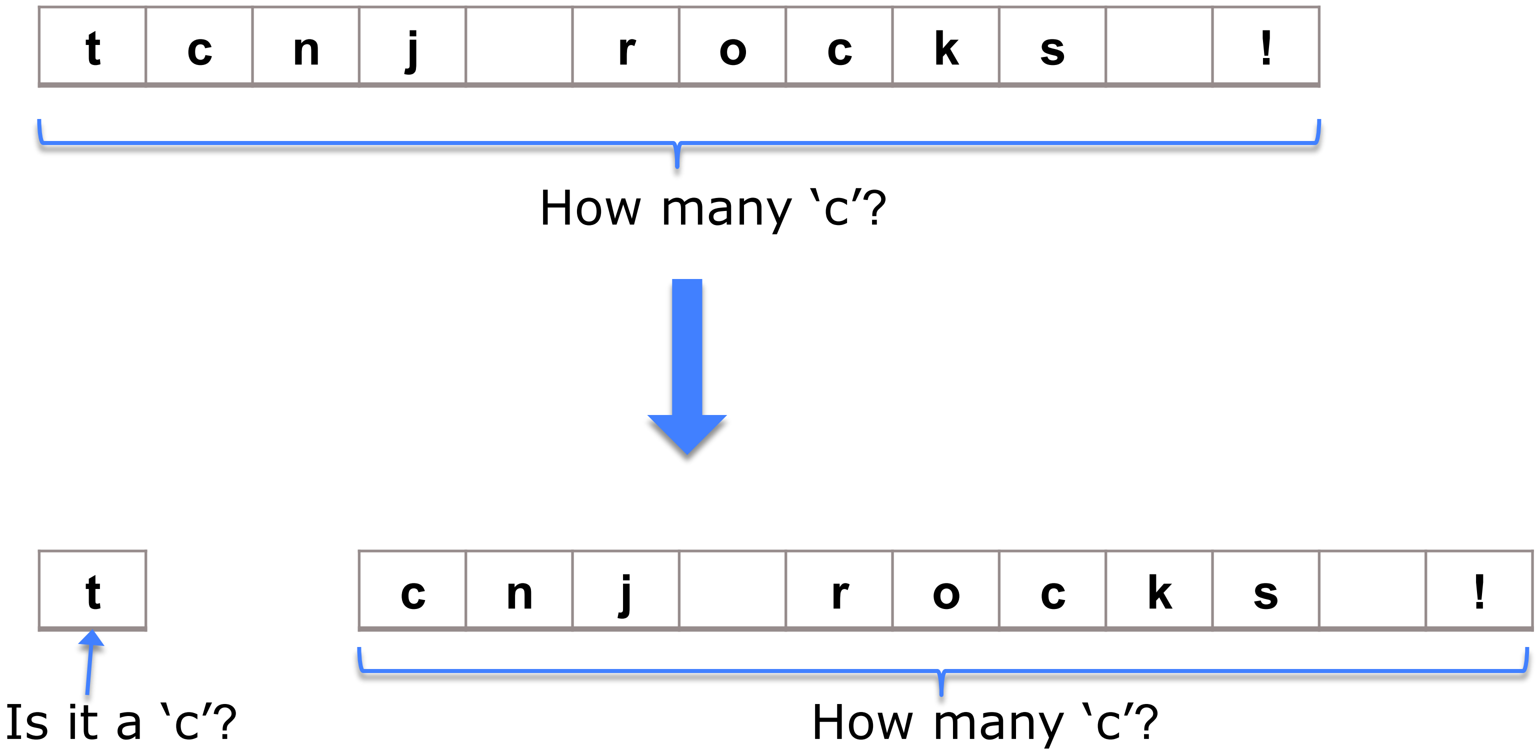
```
#include <iostream>
#include <string>
using namespace std;

/* = "s is a palindrome" */
bool isPal(string s) {
    if (s.length() <= 1) // base case
    return true;

    // { s has at least 2 chars }
    int n= s.length();
    return s.at(0) == s.at(n-1)  &&  isPal(s.substr(1, n-2));
}

int main(){
    cout<<isPal("abccba")<<endl;
    cout<<isPal("tcnj")<<endl;
}
```

Example: Count 'c' in a string



Example: Count 'c' in a string


```
#include <iostream>
#include <string>
using namespace std;

static int charCt(char c, string s){
  if(s.length() == 0) return 0;

  if(s.at(0) != c)
    return charCt(c, s.substr(1));
  else
    return 1 + charCt(c, s.substr(1));
}

int main(){
  cout << charCt('c', "tcnj rocks!") << endl;
}
```

Substring s[1..], i.e. s[1], …, s(s.length()-1)

charCt('c', "tcnj rocks") = 2

charCt('e', "new jersey") = 3

Example: The Factorial Function (n!)

- Define **$n!$** $= n\cdot(n-1)\cdot(n-2)\cdots3\cdot2\cdot1$
  read: "n factorial"
  E.g. $3! = 3\cdot2\cdot1 = 6$

- Looking at definition, can see that $n! = n * (n-1)!$

- By convention, $0! = 1$

- The function int → int that gives n! on input n is called the factorial function

A Recursive Program

$0! = 1$

$n! = n \cdot (n-1)!, \quad n > 0$

```
/* = n!. Precondition: n >= 0 */
static int fact(int n) {
   if (n = = 0)
        return 1;
   // { n > 0 }
   return n*fact(n-1);
}
```

Approach to Writing Recursive Functions

1. Find *base case(s)* – small values of n for which you can just write down the solution (e.g. 0! = 1)

2. Try to find a parameter, say n, such that the solution for n can be obtained by combining solutions to the ***same problem*** *using* ***smaller*** *values of n* (e.g. (n-1) in our factorial example)

3. Verify that, for any valid value of n, applying the reduction of step 1 repeatedly will ultimately hit one of the base cases

The Fibonacci Function

Mathematical definition:

fib(0) = 0

fib(1) = 1

two base cases!

fib(n) = fib(n − 1) + fib(n − 2), $n \geq 2$

Fibonacci sequence: 0, 1, 1, 2, 3, 5, 8, 13, …

```
/** = fibonacci(n). Pre: n >= 0 */
static int fib(int n) {
  if (n <= 1) return n;
  // n > 1
  return fib(n-2) + fib(n-1);
}
```

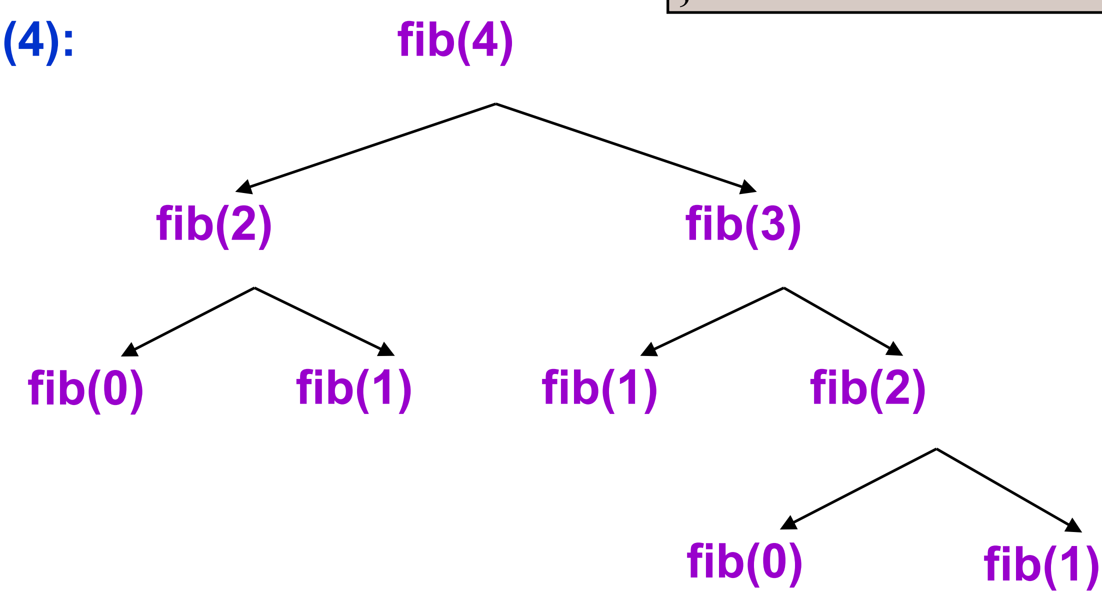
Fibonacci (Leonardo Pisano)
1170-1240?

Statue in Pisa, Italy
Giovanni Paganucci
1863

Recursive Execution


```
/* = fibonacci(n) …*/
static int fib(int n) {
  if (n <= 1) return n;
  // { 1 < n }
  return fib(n-2) + fib(n-1);
}
```

Execution of fib(4):



Comparison: Recursion v.s. Iteration

- How to compute Fibonacci Function with iteration?
- Example code
- Which one is more efficient ? Iteration or Recursion?

- According to the memory management lectures, how will the recursion and iteration be executed in memory?

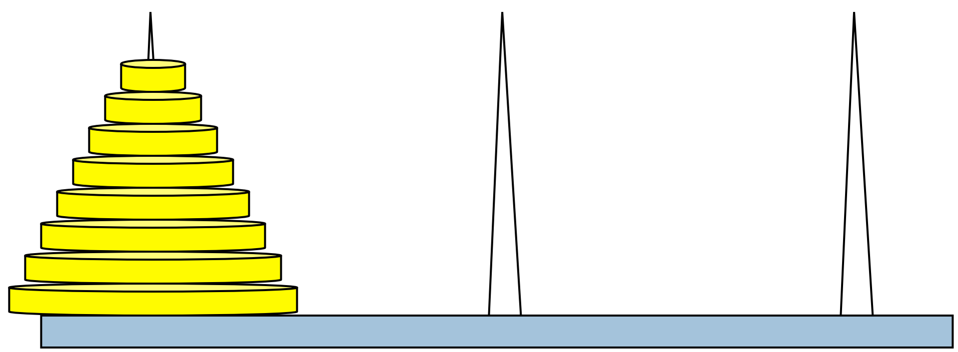
Comparison: Recursion v.s. Iteration

- Roughly speaking, recursion and iteration perform the same kinds of tasks:
  - Solve a complicated task one piece at a time, and combine the results
- Emphasis of iteration:
  - keep repeating until a task is “done” e.g., loop counter reaches limit, linked list reaches null pointer, instream.eof() becomes true
- Emphasis of recursion:
  - Solve a large problem by breaking it up into smaller and smaller pieces until you can solve it; combine the results.

  - Recursion is usually simpler to implement and easy to follow.

Example: Tower of Hanoi

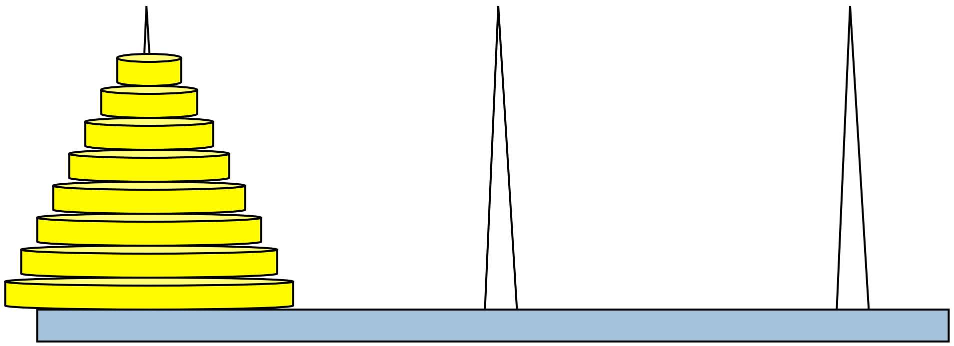
Legend has it that there were three diamond needles set into the floor of the temple of Brahma in Hanoi.



- Stacked upon the leftmost needle were 64 golden disks
- each a different size
- stacked in concentric order

A Legend

The monks were to transfer the disks from the first needle to the second needle, using the third as necessary.

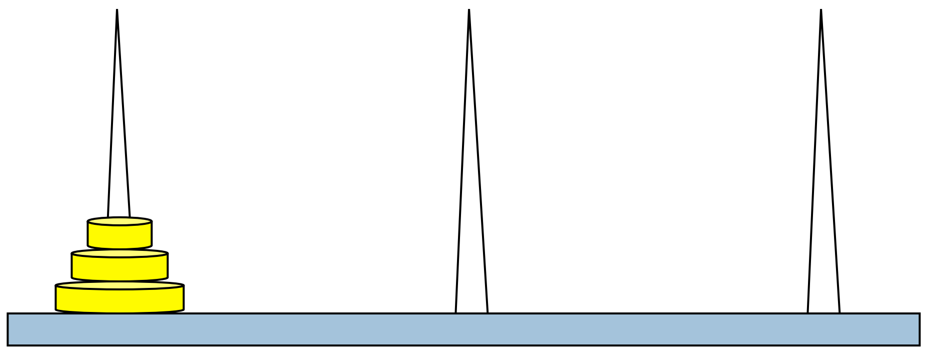


- *move one disk at a time*
- could *never put a larger disk on top of a smaller one*.

When they complete this task, will the world end?

To Illustrate

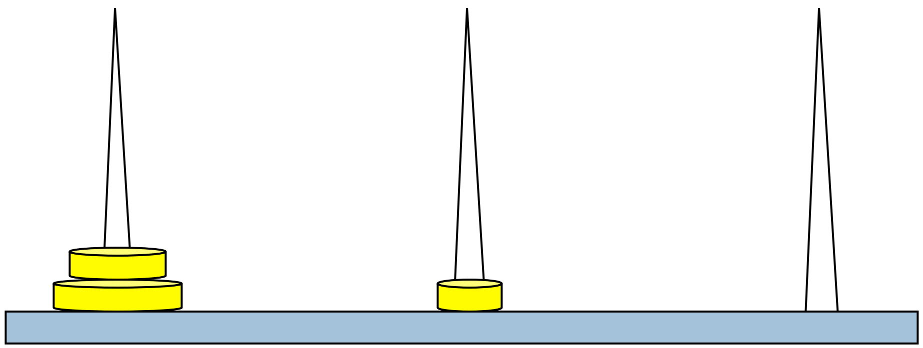
For simplicity, suppose there were just 3 disks, and we' ll refer to the three needles as A, B, and C...



Since we can only move one disk at a time, we move the top disk from A to B.

Example

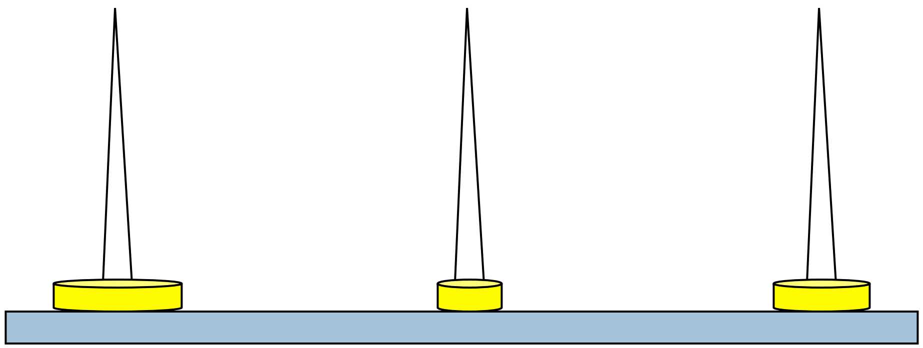
For simplicity, suppose there were just 3 disks, and we' ll refer to the three needles as A, B, and C...



We then move the top disk from A to C.

Example (Ct’ d)

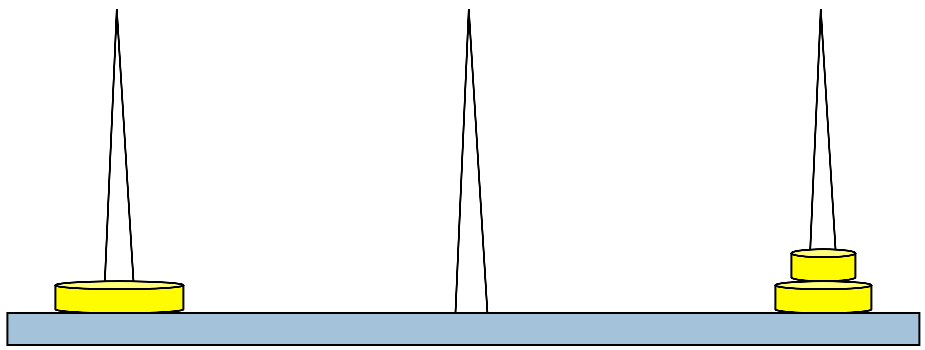
For simplicity, suppose there were just 3 disks, and we’ ll refer to the three needles as A, B, and C...



We then move the top disk from B to C.

Example (Ct’ d)

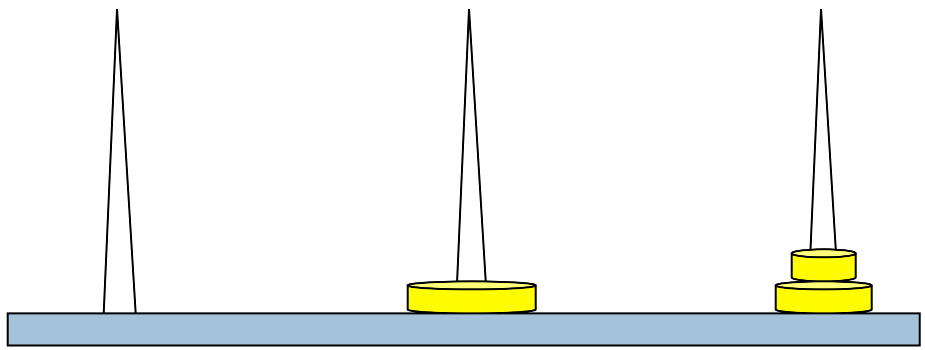
For simplicity, suppose there were just 3 disks, and we’ ll refer to the three needles as A, B, and C...



We then move the top disk from A to B.

Example (Ct’ d)

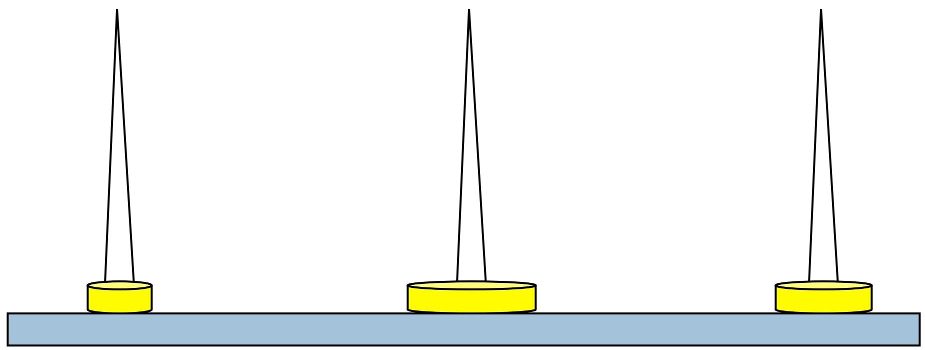
For simplicity, suppose there were just 3 disks, and we’ ll refer to the three needles as A, B, and C...



We then move the top disk from C to A.

Example (Ct’ d)

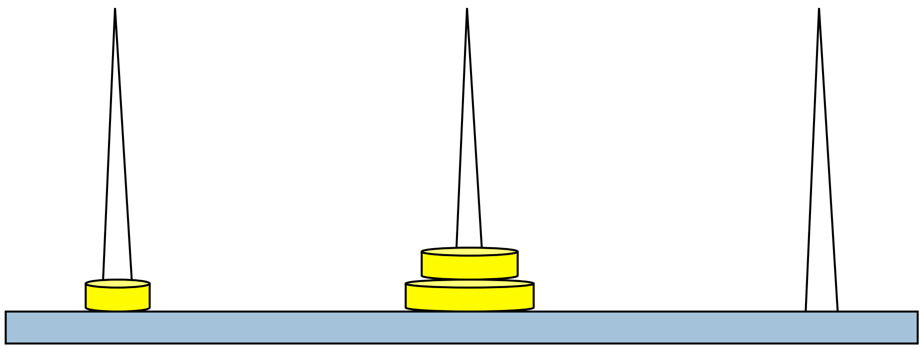
For simplicity, suppose there were just 3 disks, and we’ ll refer to the three needles as A, B, and C...



We then move the top disk from C to B.

Example (Ct’ d)

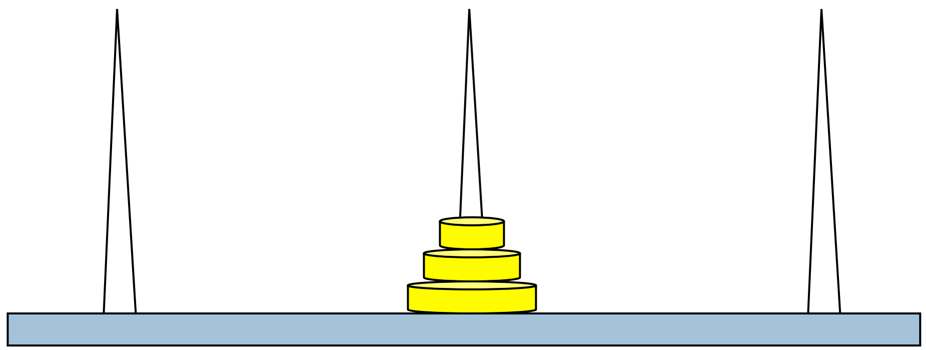
For simplicity, suppose there were just 3 disks, and we’ ll refer to the three needles as A, B, and C...



We then move the top disk from A to B.

Example (Ct’ d)

For simplicity, suppose there were just 3 disks, and we’ ll refer to the three needles as A, B, and C...

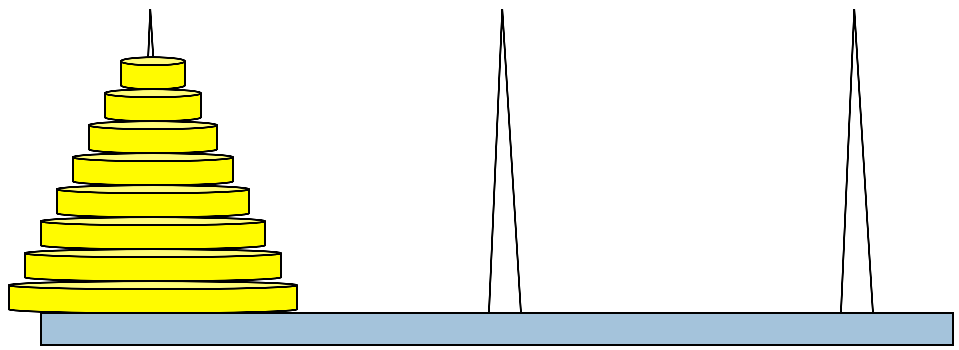


and we’re **done**!

The problem gets more difficult as the number of disks increases...

Our Problem

Today' s problem is to write a program that generates the instructions for the priests to follow in moving the disks.



While quite difficult to solve iteratively, this problem has a simple and elegant ***recursive*** solution.

General Approach to Writing Recursive Functions

1. Find *base case(s)* – small values of n for which you can just write down the solution (e.g. 0! = 1)

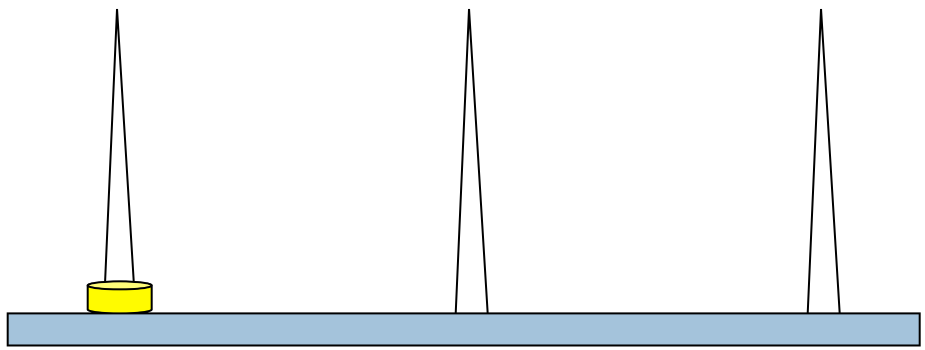
2. Try to find a parameter, say n, such that the solution for n can be obtained by combining solutions to the ***same problem*** *using* ***smaller*** *values of n* (e.g. (n-1) in our factorial example)

3. Verify that, for any valid value of n, applying the reduction of step 1 repeatedly will ultimately hit one of the base cases

Design

Basis: What is an instance of the problem that is trivial?

→ n == 1

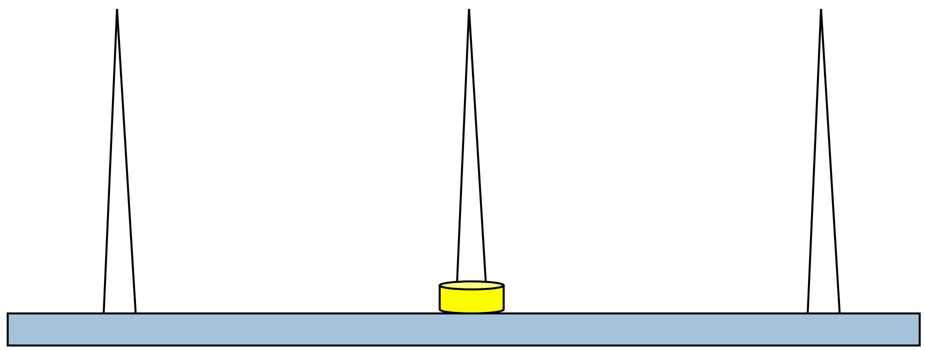


Since this base case could occur when the disk is on any needle, we simply output the instruction to move the top disk from *A* to *B*.

Design

Basis: What is an instance of the problem that is trivial?

→ n == 1

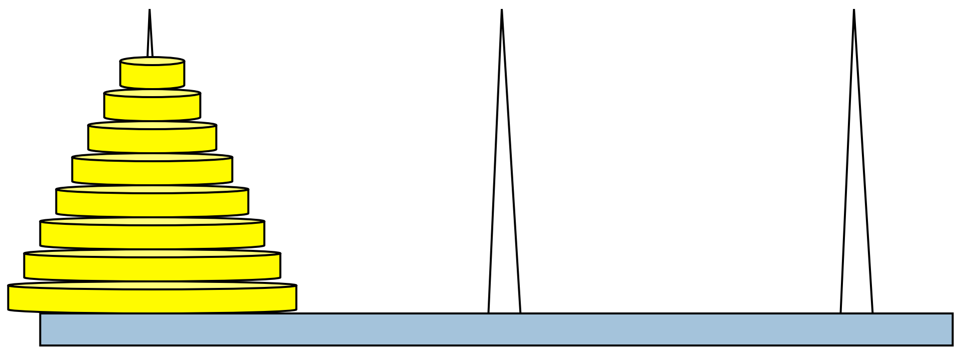


Since this base case could occur when the disk is on any needle, we simply output the instruction to move the top disk from *A* to *B*.

Design (Ct’ d)

Induction Step: n > 1

→ How can recursion help us out?

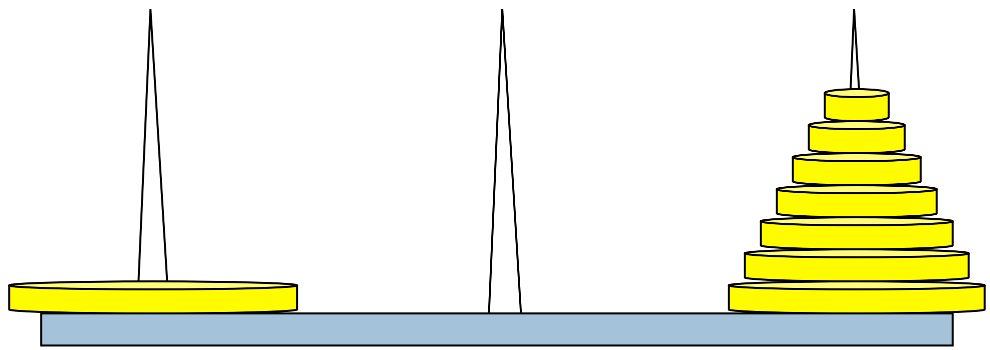


a. <u>Recursively</u> move n-1 disks from *A* to *C*.

Design (Ct’d)

Induction Step: n > 1

→ How can recursion help us out?

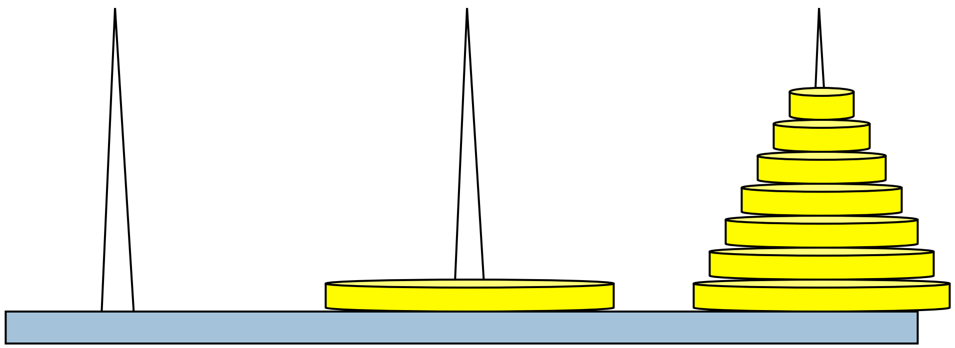


b. Move the one remaining disk from *A* to *B*.

Design (Ct’ d)

Induction Step: n > 1

→ How can recursion help us out?

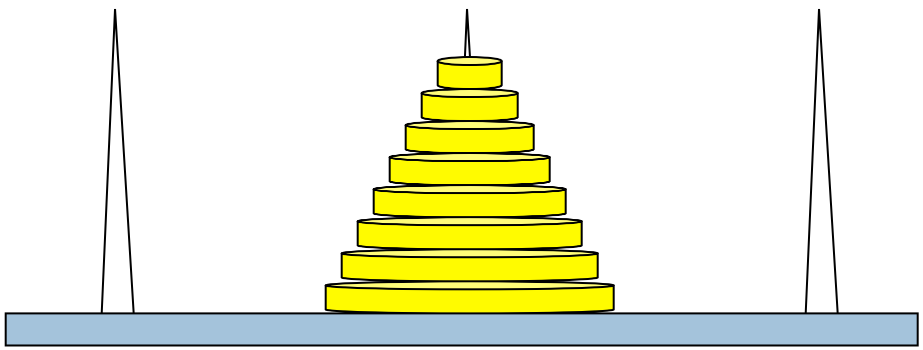


c. *Recursively* move n-1 disks from *C* to *B*...

Design (Ct’d)

Induction Step: n > 1

→ How can recursion help us out?



d. We’re done!

Tower of Hanoi: Code


```
void Hanoi(int n, string a, string b, string c)
  {
     if (n == 1)  /* base case */
        Move(n, a, b);  // move disk n from a to b
     else { /* reduction */
        Hanoi(n-1,a,c,b);
        Move(n, a, b);
        Hanoi(n-1,c,b,a);
     }
  }
```

Non-Negative Integer Powers

$a^n = a \cdot a \cdot a \cdots a$ (n times)

Alternative description:

- $a^0 = 1$
- $a^{n+1} = a \cdot a^n$

```
/* = a^n. Pre: n >= 0 */
static int power(int a, int n) {
   if (n == 0) return 1;
   return a*power(a, n-1);
}
```

A Smarter Version

Power computation:

- $a^0 = 1$
- If n is **nonzero** and **even**, $a^n = (a*a)^{n/2}$
- If n is **nonzero**, $a^n = a * a^{n-1}$

C++ note: For ints x and y, x/y is the integer part of the quotient

Judicious use of the second property makes this a logarithmic algorithm, as we will see

Example: $3^8 = (3*3) * (3*3) * (3*3) * (3*3) = (3*3)^4$

Smarter Version in C++

- $n = 0$: $a^0 = 1$
- n nonzero and even: $a^n = (a*a)^{n/2}$
- n nonzero: $a^n = a \cdot a^{n-1}$

```
/* = a**n. Precondition: n >= 0 */
static int power(int a, int n) {
  if (n == 0) return 1;
  if (n%2 == 0) return power(a*a, n/2);
  return a * power(a, n-1);
}
```

Build table of multiplications

n	n	mults
0		0
1	2^0	1
2	2^1	2
3		3
4	2^2	3
5		4
6		4
7		4
8	2^3	4
9		5
…		
16	2^4	5

Start with n = 0, then n = 1, etc. For each, calculate number of mults based on method body and recursion.

See from the table: For n a power of 2, $n = 2^k$, only $k+1 = (\log n) + 1$ mults

For $n = 2^{15} = 32768$, only 16 mults!

```
static int power(int a, int n) {
    if (n == 0) return 1;
    if (n%2 == 0) return power (a*a, n/2);
    return a * power (a, n-1);
}
```

How C++ "compiles" recursive code

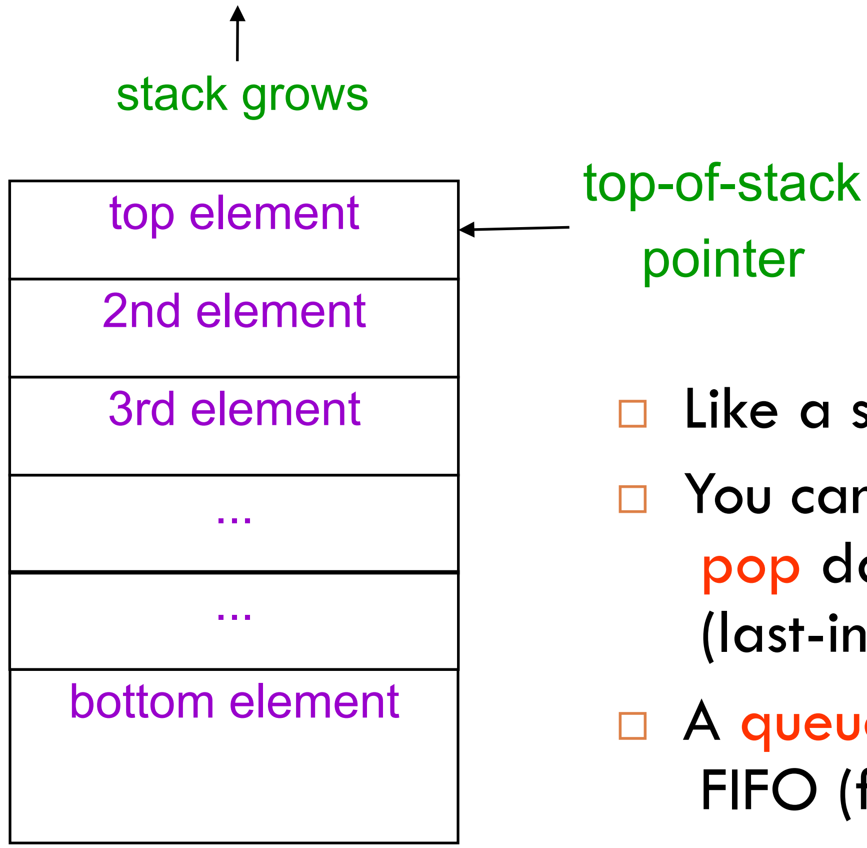
Key idea:

- C++ uses a stack to remember parameters and local variables across recursive calls
- Each function invocation gets its own stack frame

A stack frame contains storage for

- Local variables of method
- Parameters of method
- Return info (return address and return value)
- Perhaps other bookkeeping info

Stacks



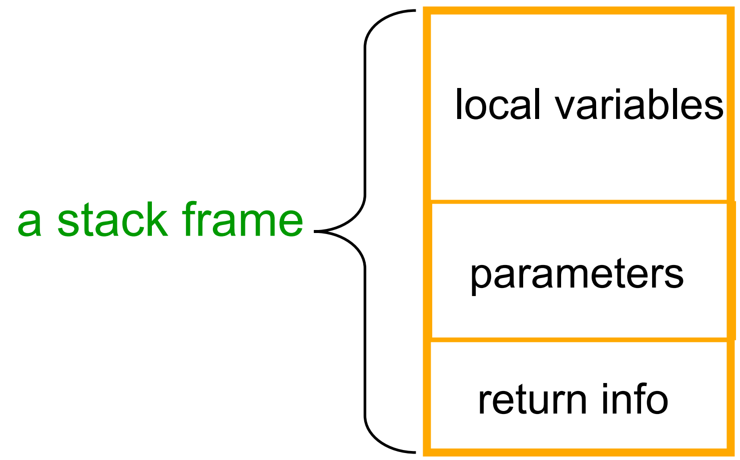
- Like a stack of dinner plates
- You can push data on **top** or pop data off the **top** in a **LIFO** (last-in-first-out) fashion
- A queue is similar, except it is FIFO (first-in-first-out)

Stack Frame

A new stack frame is pushed with each **function call**

The stack frame is **popped** when the function returns

- Leaving a return value (if there is one) on top of the stack

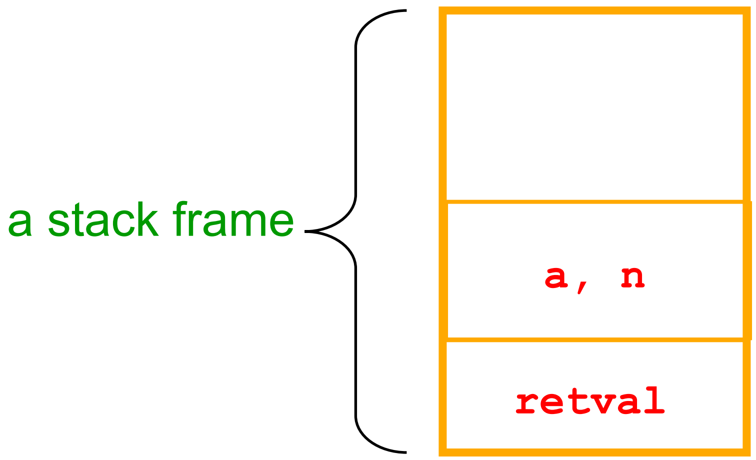


Stack Frame

A new stack frame is pushed with each **function call**

The stack frame is **popped** when the function returns

- Leaving a return value (if there is one) on top of the stack



How Do We Keep Track?

- Many frames may exist, but computation occurs **only** in the top **frame**
  - The ones below it are waiting for results

- The hardware has nice support for this way of implementing function calls, and recursion is just a kind of function call

Conclusion

Recursion is a convenient and powerful way to define functions

Problems that seem insurmountable can often be solved in a "divide-and-conquer" fashion:

- Reduce a big problem to smaller problems of the same kind, solve the smaller problems
- Recombine the solutions to smaller problems to form solution for big problem

Lab 6 discussion

- Flexible “loop” using recursion

- How to loop from 000 to 999 with FOR loop
  - for
    - for
      - for

- How to build it with recursion?
- com(int len, int size)
- len: the number of elements that needs to loop
- size: the total length of the output string
- demo

Project 2

- What is the value of Project 2?
- Why we want to implement it with arrays?
- What should we concern about this project?