

CS213 – 2022/ 2023

Programming II

Lecture 15: Recursion

By

Dr. Mohammad El-Ramly

- <https://web.itu.edu.tr/~bkurt/Courses/blg332ss/>

# Course Organization

- This week
  - We will cover recursion
  - And backtracking
- Next weeks
  - STL / Templates
  - Exceptions
  - .....

# Obfuscated C Contest

• كيف تكتب أبشع الأكواد لعمل أبسط الأشياء

- <https://www.ioccc.org/>
- See an example  
<https://www.ioccc.org/2018/endoh1/prog.c>  
<https://www.ioccc.org/2018/algmyr/prog.c>

# صدق أو لا تصدق

## هذا الهراء برنامج يشتغل

```
#define/*__Int3rn^ti[]n/l_()I3fusc^t3|]_C_C<>I7E_C[]nt3st__*/L/*__MMXVIII__*/for
#include/*!""()*+,-./12357;<=>?CEFGHIJKLMNOPSTUVWXYZ[]^_`cfghijklmnrstuvwxyz{|}*/<stdio.h>
char*r,F[1<<21]="~T/}3(|+G{>/zUhy;Jx+5wG<v>>u55t.?sIZrC]n.;m+:l+Hk]WjNji/Sh+2f1>c2H`)(_2(^L\
-]=([1/Z<2Y7/X12W:.VFFU1,T77S+;N?;M/>L..K1+JCCI<<H:(G*5F--E11C=5?.(>+(=3)Z-;*(:.Y/5(-=)2*-U,\
/+~?5'(+,+++***'EE>T,215IEUF:N`2`?:GK;+^+?>)5?>U>_)5GxG).2K.2};}_235():5,S7E1(vTSS,-SSTvU(<-HG\
-2E2/2L2/EE->E:EE,2XMMM1Hy`)5rHK;+.T+?[n2/_2{LKN2/_|cK2+.2`;}:?{KL57?|cK:2{NrHKtMMMK2nrH;rH[n"
"CkM_E21-E,-1->E(_:mSE/LhLE/mm:2U1;2M>,2KW-+. -u).5Lm?fm`2`2nZXjj?[n<YcK?2}yC}H[^7N7LX^7N7UN</:-\
ZWXI<^I2K?>T+?KH~-?f<;G_x2;;2XT7LXIuuVF2X(G(GVV--:-:KjJ]HKLyN7UjJ3.WXjNI2KN<1|cKt2~[IsHfI2w{[<VW"
"GIfZG>x#&&&$#$;ZXIc###$&$&$>7[LMv{&&&&#&#L,12TY.&$&$&$&$,(iiii,#&&&#&$&$?TY2.$&$1(x###;2EE[t,\
SSEz.SW-k,T&jC?E-.$##&&&57+$&$&&&W1-&$&$7W-J$&$kEN&&&###C^+$&#W,h###n/+L2YE"
"2nJk/H;YNs#$[, :TU(#$ ,: &&~H>&# Y; &&G_x#2; ,mT&$YE-&# 5G $#VVF$&#zNs$&$&Ej]HELy\
CN/U^Jk71<(#&:G7E+^&# 1|?1 $$Y.2$$ 7lzs WzZw>&$E -<V-wE(2$$ G>x; 2zsW/$$#HKt&$&$v>+t1(>"
"7>S7S,;TT,&$;S7S>7&#>E_: :U $$' ",op ,*G= F,*I=957+F ;int*t,k,O, i, j,T[+060<<+020];int M(
int m,int nop){;;;return+ m%(0+nop );;} int*tOo,w, h,z,W;void(C) (int n){n=putchar(n);}int
f,c,H=11,Y=64<<2,Z,pq,X ;void(E/*d */)( int/*RP*/n ){L(Z=k+00; Z; Z/=+2+000)G[000]=*G*!!f
|M(n,2)<<f,pq=2,f+=06 <f?++pq,++pq ,G++ ,z:f+001,n /=2;;}void (V)( int/*opqrstabd*/n){C(n
%Y);;C(n/Y+00);;}void J(){L(pq--,pq =j =0=-1+0;+ j<240;I[6+ (h +6+j/12/2*2+M(j/2,2))*
W+M(j/2/2,+06)*2+w*014 +00+M(00+ 000+j,002 +00)]=000 +00+k)k=M(G[j/2/2+(*r-+
32)**"<nopqabdeg"] ,/*4649&96#*/3);/*&oaogoqo*/;}/*xD%P$Q#Rq*/int/*dbqpdbqpxyzyboo35700Q*/main()
{L(X=Y-1;i<21*3;i++,I++)L(r=G,G+=2;*G++;)*G>=13*3?*G-*r?*I++=*G:(*I++=r[1],*I++=r[2]):1;L(j=12,r
=I;(*I=i=getchar())>-1;I++)i-7-3?I-=i<32||127<=i,j+=12:(H+=17+3,W=W<j?j:W,j=12);L(;*r>-1;r++)*r-
7-3?J(),w++:(w=z,h+=17+3);C(71);C(73);V(''*'1'*7);C(57);C(32*3+1);V(W);V(H);C(122*2);L(V(i=z);i
<32*3);C(i++/3*X/31);C(33);C(X);C(11);L(G="SJYXHfUJ735";*G);C(*G++-5);C(3);V(1);L(V(j=z);j<21*3;
j++){k=257;V(63777);V(k<<2);V(M(j,32)?11:511);V(z);C(22*2);V(i=f=z);V(z);V(W);V(H);V(1<<11);r=
G=I+W*H;L(t=T;i<1<<21;i++)T[i]=i<Y?i:-1;E(Y);L(i=-1;+i<W*H;t=T+Z*Y+Y)c=I[i]?I[i]*31-31:(31<
j?j-31:31-j),Z=c[t[c]<z?E(Z),k<(1<<12)-2?t[c]=++k,T:T:t];E(Z);E(257);L(G++;k=G-r>X?X:G-r
,C(k),k);)L(;k--;C(*r++/*---#&$&04689@ABDOPQRabdegopq---*/));}C(53+6);return(z);}
```

الشتاء ربيع المؤمن  
قصر نهاره فصامه و طال ليله فقامه

”بئس مطية الرجل زعموا“ حديث شريف

# Lecture Goals

- Learn recursive thinking and problem solving.
- Working out some recursive solutions for some problems
  1. Factorial
  2. Palindrome
  3. Tower of Hanoi
  4. Binary Search
  5. Fibonacci Numbers

# Problem Solving Strategies

- **Iteration** is repeating the same task a number of times or until a condition is not valid anymore.
  - `for`, `while`, etc.
- **Recursion** is solving a problem by reducing it to a smaller problem of the same form.



# Recursion

- An algorithm is recursive if it calls itself to do part of its work.
- Example:
  1. Compute  $n!$
  2. Hanoi puzzle
  3. Merge sort
  4. Binary search



# Recursion

- Recursion helps reducing the complexity of the problem.
- A recursive solution will require
  - A **recursive case**, in which a small version of the sub-problem is solved, and
  - A **base case** in which recursion stops and solving the smallest version of the problem gives a fixed value.



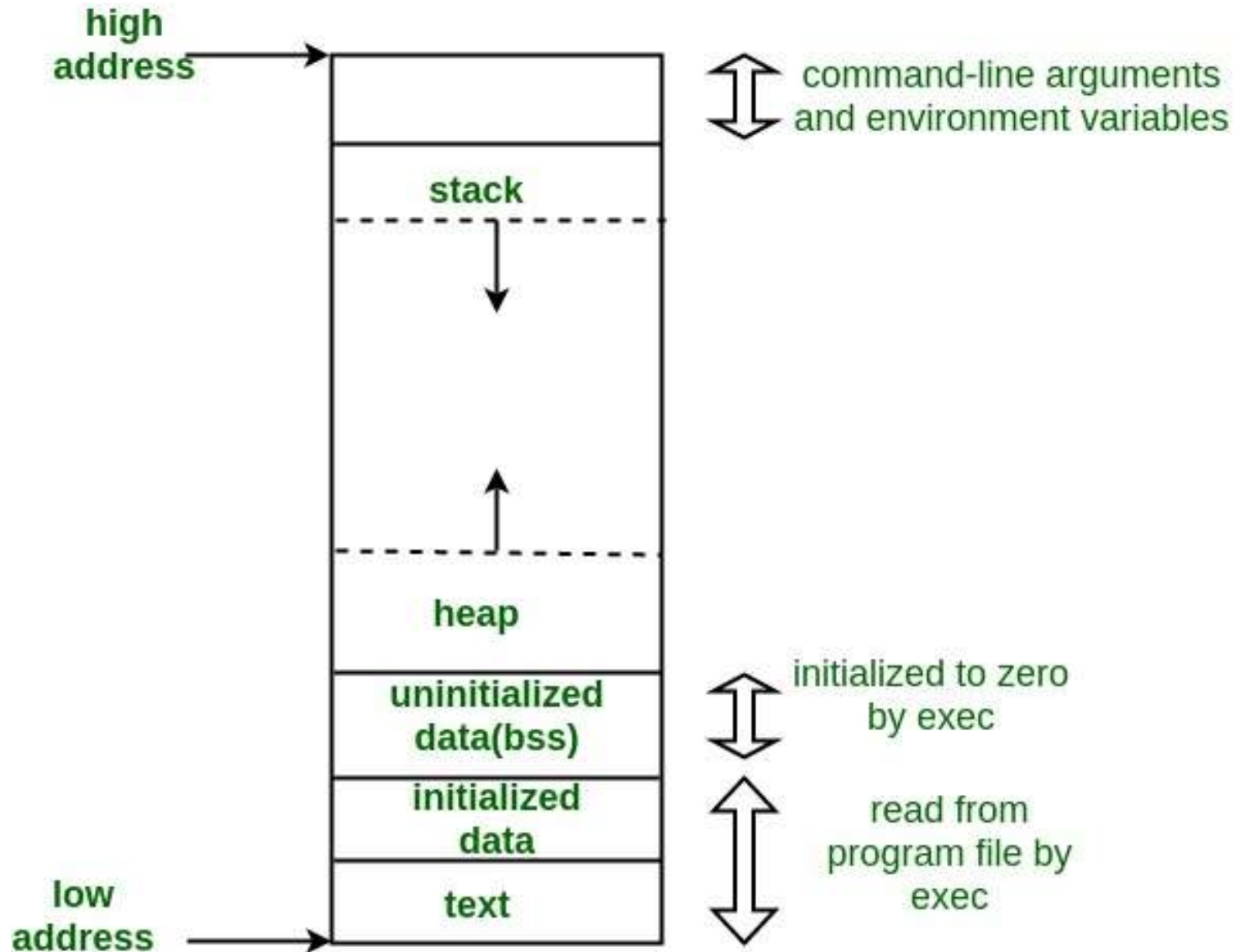
# Recursion in Programming

- We have been calling other methods from a method.
- It's also possible for a method to call itself.
- A method that calls itself is a *recursive method*.
- Such a call will solve the same problem but with different (reduced) parameters.

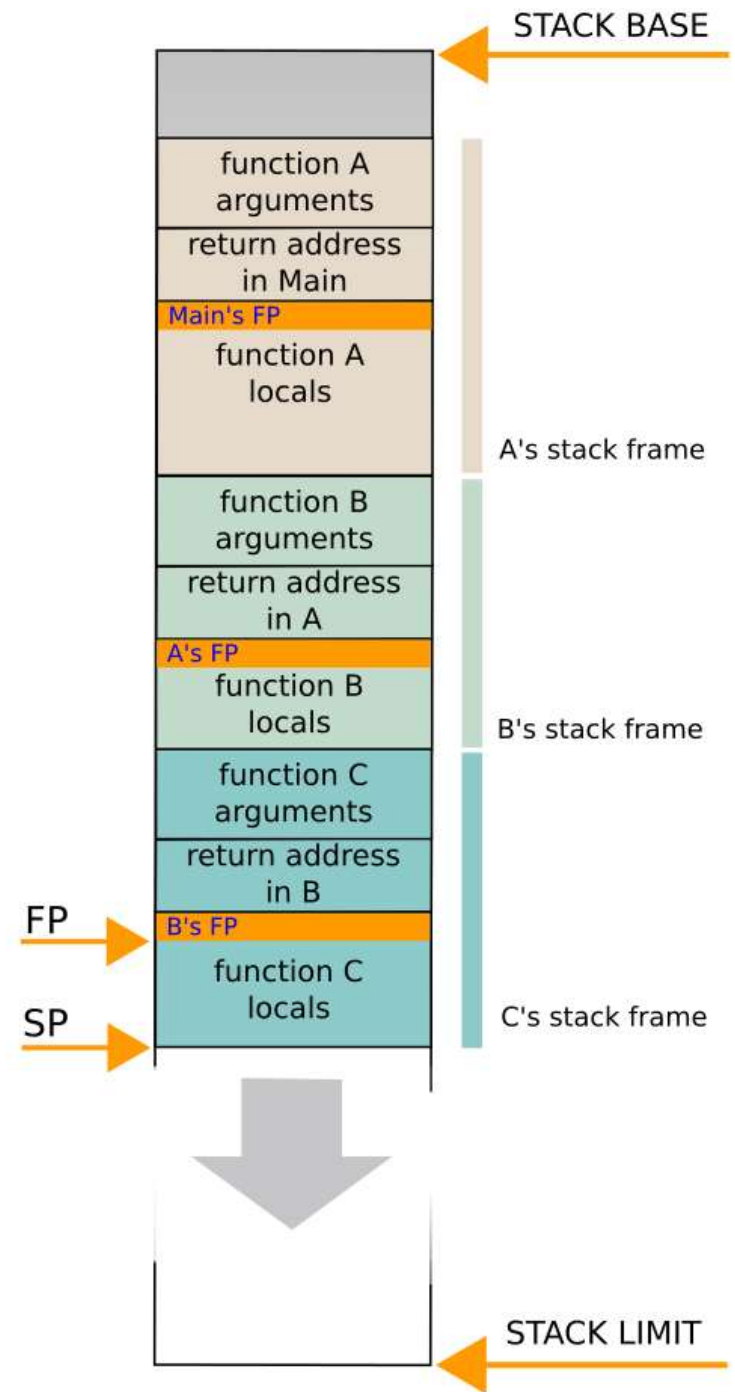
# Solving Problems With Recursion

- Recursion can be a powerful tool for solving repetitive problems.
- Recursion is never absolutely required to solve a problem.
- Any problem that can be solved recursively can also be solved iteratively, with a loop.
- In many cases, recursive algorithms are **less efficient** than iterative algorithms.

# C++ Memory Layout



# C++ Stack Layout



# Solving Problems With Recursion

- Recursive solutions repetitively:
  - **allocate memory** for **parameters** and **local variables**, and
  - **store the address** of where control returns after the method terminates.
- These actions are called ***overhead*** and take place with each method call.
- This overhead does not occur with a **loop**.
- Some repetitive problems are more easily solved with recursion than with iteration.
  - **Iterative** algorithms might **execute faster**; however,
  - a **recursive** algorithm might be **designed faster**.





# 1. Calculating Factorial

- The factorial of a nonnegative number can be defined by the following rules:
  - If  $n = 0$  then  $n! = 1$
  - If  $n > 0$  then  $n! = 1 \times 2 \times 3 \times \dots \times n$

# Iterative Factorial

```
int Fact(int n) {  
    int product;  
    product = 1;  
    for (int i = 1; i <= n; i++)  
        product *= i;  
    return product;  
}
```

# Using Recursion

- The iterative implementation of **Fact**, however, does not take advantage of an important mathematical property of factorials:

$$n! = \begin{cases} 1 & \text{if } n = 0 \\ n \times (n - 1)! & \text{otherwise} \end{cases}$$

- $4! = 4 * 3! = 4 * 3 * 2! = 4 * 3 * 2 * 1 = 24$

# Recursive Factorial

- So, we can see that:
  - when  $n$  is 0, its factorial is 1, and
  - when  $n$  greater than 0, its factorial is the product of all the positive integers from 1 up to  $n$ .
- Factorial(6) is calculated as
  - $1 \times 2 \times 3 \times 4 \times 5 \times 6$ .
- **The base case** is where  $n$  is equal to 0:  
if  $n = 0$  then  $\text{factorial}(n) = 1$
- **The recursive case**, or the part of the problem that we use recursion to solve is:
  - *if  $n > 0$  then  $\text{factorial}(n) = n \times \text{factorial}(n - 1)$*

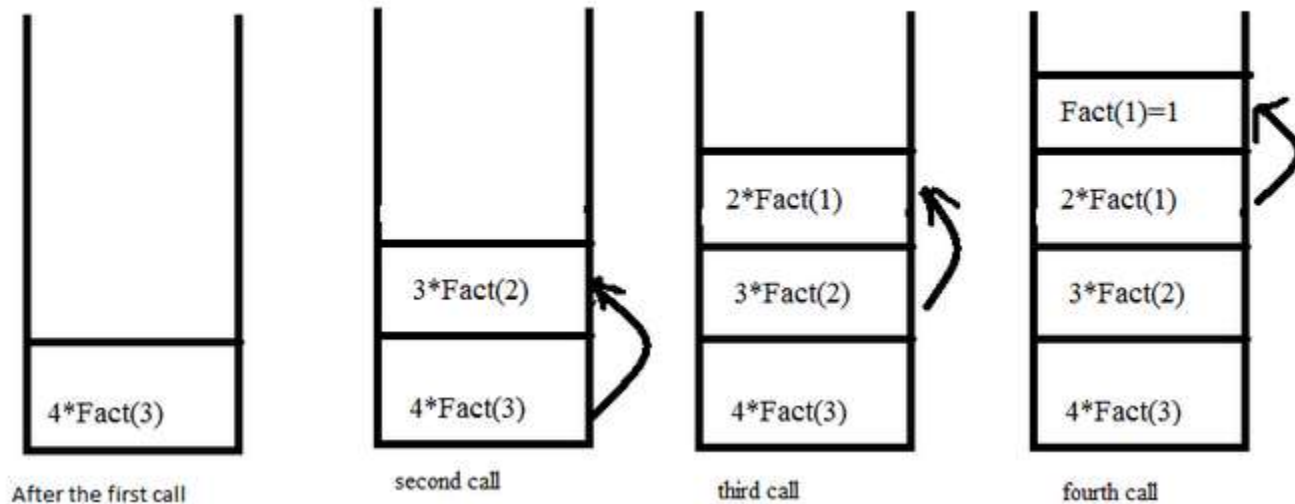
# Recursive Factorial

```
int RecursiveFact(int n) {  
    if (n <= 1)  
        return 1;  
    else  
        return n * RecursiveFact (n-1);  
}
```

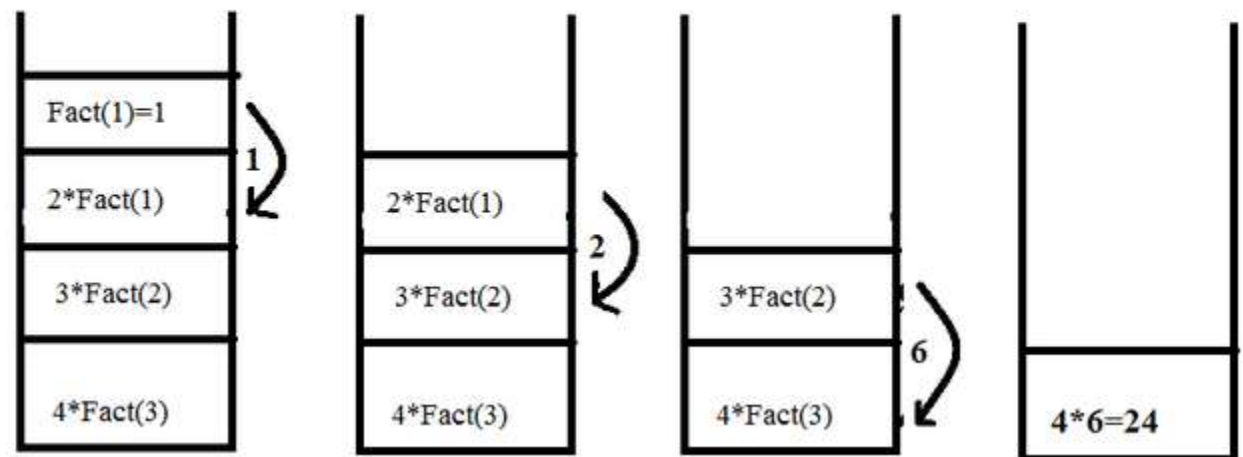
```
int RecursiveFact(int n) {  
    return n <= 1 ? 1 : n * RecursiveFact (n-1);  
} // Not recommended - hard to follow
```

# C++ Stack for Recursive Factorial

When function call happens previous variables gets stored in stack



Returning values from base case to caller function





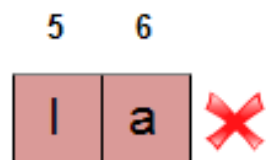
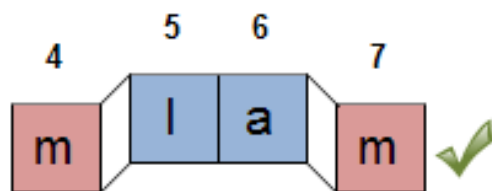
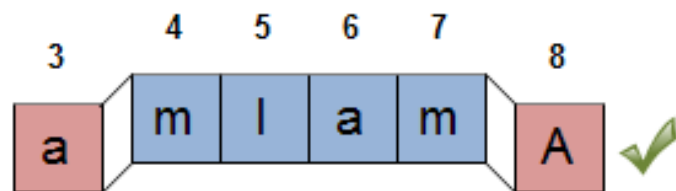
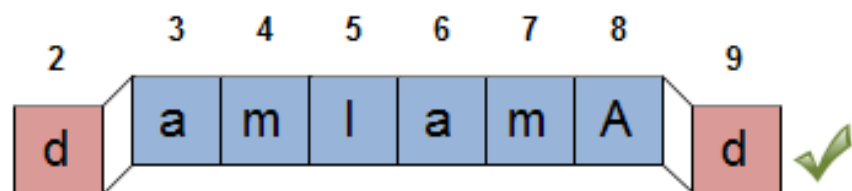
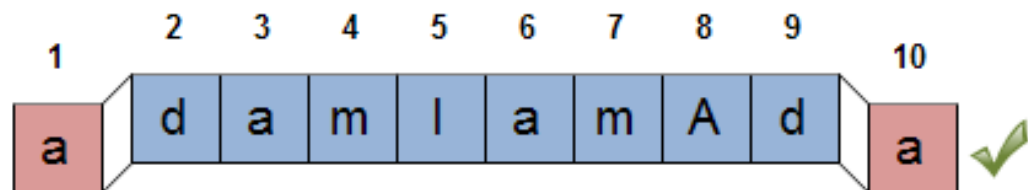
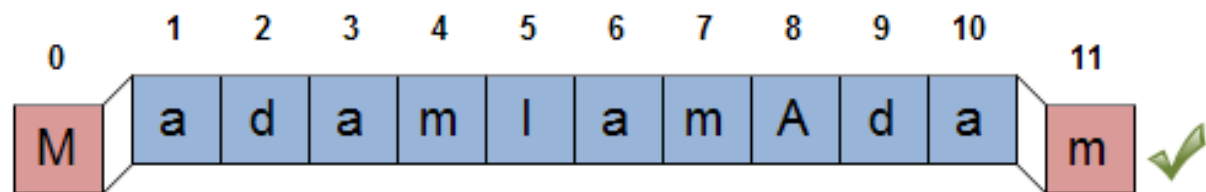
## 2. Exercise: Detecting Palindromes

- A **palindrome** is a string that reads identically backward and forward, such as "level", "noon".
  - Can you **detect palindromes recursively**?
    - Any palindrome longer than a single character must contain a shorter palindrome in its interior.
    - The string "level" consists of the palindrome "eve" with an "l" at each end.
- 1. Base case:** a string with 0 or 1 characters is a palindrome.
  - 2. Recursive case:** a string is a palindrome if 1<sup>st</sup> and last characters are equal and substring resulting from removing them is a palindrome.



# Exercise: Detecting Palindromes

```
bool IsPalindrome(string str) {  
    int len = str.length();  
    if (len <= 1)  
        return true;  
    else  
        return (str[0] == str[len - 1]  
            && IsPalindrome(str.substr(1, len - 2)));  
}
```



# Exercise: Detecting Palindromes

- Can you make it more efficient. Why?
  1. Calculate the length of the argument string only once.
  2. Don't make a substring on each call.
    - Calculate length once and pass pointers or indices of the currently active substring.

# Exercise: Detecting Palindromes

```
bool CheckPalindrome
(string str, int start, int end) {
    if (end <= start) return true;
    else return str[start] == str[end]
        && CheckPalindrome (str, ++start, --end);
}

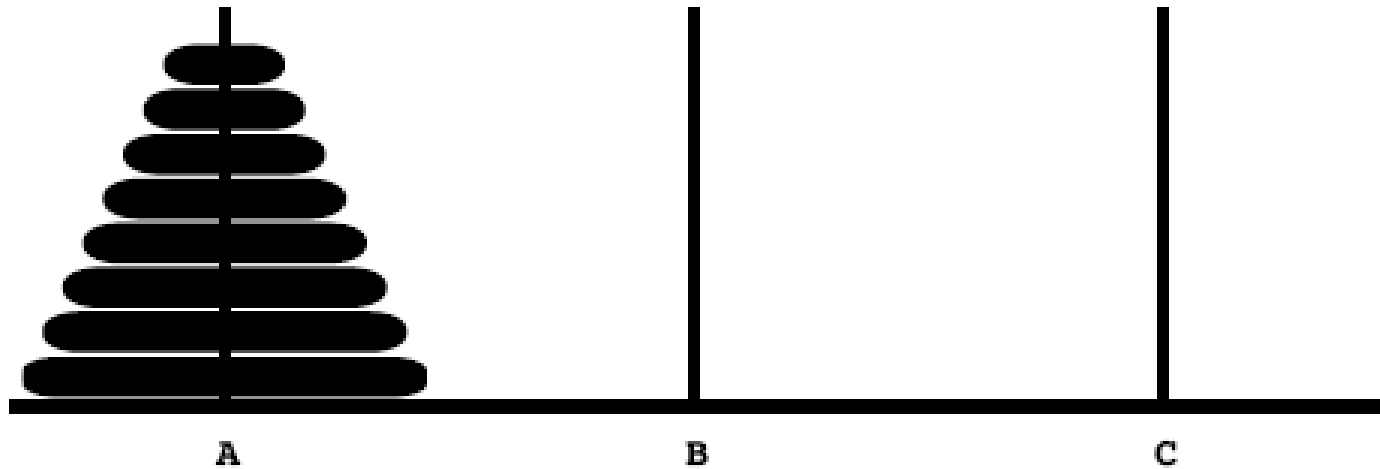
// Wrapper class to keep same interface
bool IsPalindrome(string str) {
    return CheckPalindrome
        (str, 0, str.length()-1);
}
```



# 3. Tower of Hanoi

- Tower of Hanoi is a simple puzzle invented by French mathematician Edouard Lucas in the 1880s,
- It is a classical recursion problem.
- You have 64 stacked disks of varying sizes ordered from the largest at the bottom to the smallest on the top. The pole goes through all the poles. You need to move them from one pole to another. You have one extra intermediate pole. (1) No disk can be moved outside a pole. But it can be moved from a pole to another. (2) No large disk can be above a smaller one at anytime.
- <http://www.mazeworks.com/hanoi/index.htm>

# Tower of Hanoi



## Inputs:

1. The number of disks to move
2. The name of the starting pole
3. The name of the ending pole
4. The name of the temporary pole

# Solving Tower of Hanoi

- *Base case*

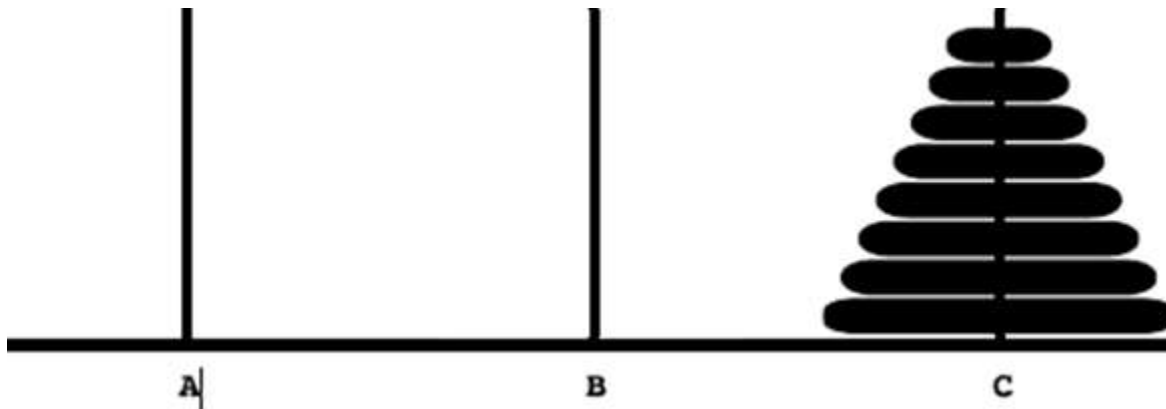
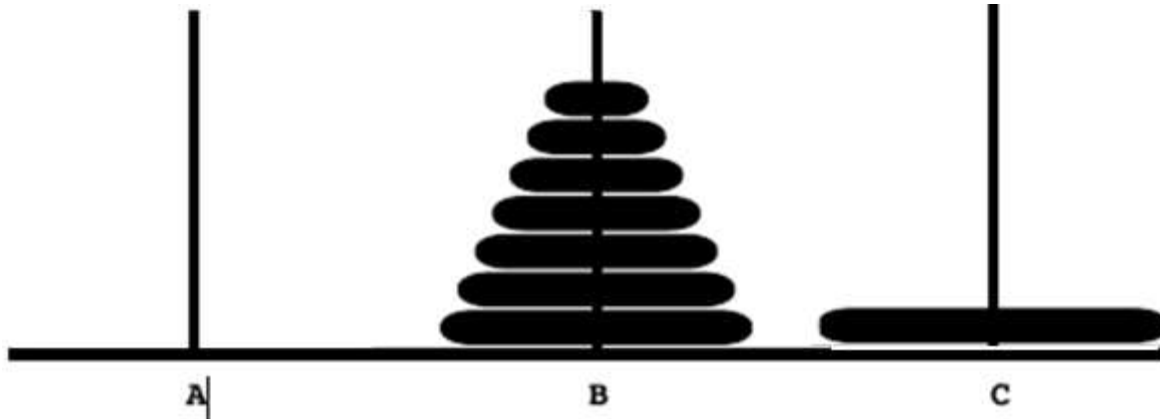
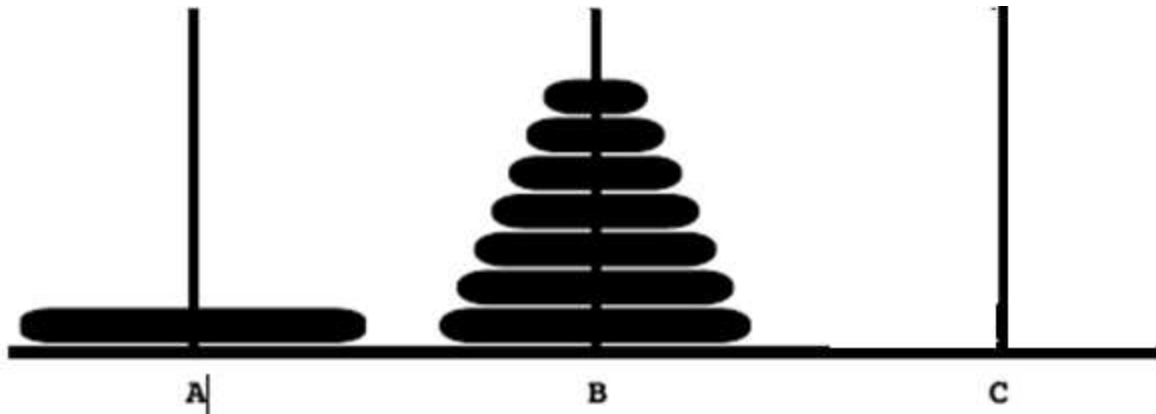
- The simple case occurs when **n is equal to 1**.
- If the tower only contains one disk, you can go ahead and move it.

- *Recursive case*

- For n disks, **move n-1** disks to the **temporary** pole and the **bottom disk n** to the **ending pole** and then move the first **n-1** from the **temporary** pole to the **ending** one.



# Tower of Hanoi



# Tower of Hanoi

```
MoveTower(int n, start, finish, temp) {  
    if (n == 1)  
        Move a single disk from start to finish.  
    else {  
        Move a tower of size n-1 from start to temp.  
        Move a single disk from start to finish.  
        Move a tower of size n-1 from temp to finish.  
    }  
}
```

# Tower of Hanoi

```
void MoveSingleDisk(char start, char finish) {
    cout << start << " -> " << finish << endl;
}

void MoveTower(int n, char start, char finish, char temp){
    if (n == 1)
        MoveSingleDisk(start, finish);
    else {
        MoveTower(n - 1, start, temp, finish);
        MoveSingleDisk(start, finish);
        MoveTower(n - 1, temp, finish, start);
    }
}

int main() {
    MoveTower(5, 'A', 'C', 'B');
}
```

- Number of Disks (n)      Number of Moves

|   |                         |
|---|-------------------------|
| 1 | $2^1 - 1 = 2 - 1 = 1$   |
| 2 | $2^2 - 1 = 4 - 1 = 3$   |
| 3 | $2^3 - 1 = 8 - 1 = 7$   |
| 4 | $2^4 - 1 = 16 - 1 = 15$ |
| 5 | $2^5 - 1 = 32 - 1 = 31$ |
- The steps to transfer n disks from post A to post B is:  $2^n - 1$ .
- Even if it only takes one second to make each move, it will be  $2^{64} - 1$  seconds before the world will end. This is 590,000,000,000 years (that's 590 billion years) - That's a really long time!

# Recursion or Iteration?

- Two ways to solve particular problem
  - Iteration
  - Recursion
- Iterative control structures: uses looping to repeat a set of statements
- Tradeoffs between two options
  - Sometimes recursive solution is easier
  - Recursive solution is often slower



# 4. Recursive Binary Search

- The binary search algorithm can be implemented recursively.
- The procedure can be expressed as:

If array diminishes (size 0), return not found.

If array[middle] equals the search value, then the value is found.

Else

if array[middle] is less than the key value, do a binary search on the upper half of the array.

Else

if array[middle] is greater than the key value, perform a binary search on the lower half of the array.

# Recursive Binary Search

```
int BinarySearch
```

```
  (string key, string array[], int low, int high) {
```

```
    if (low > high) return -1;
```

```
    int mid = (low + high) / 2;
```

```
    if (key == array[mid]) return mid;
```

```
    if (key < array[mid])
```

```
        return BinarySearch(key, array, low, mid - 1);
```

```
    else
```

```
        return BinarySearch(key, array, mid + 1, high);
```

```
}
```

```
int FindStringInSortedArray
```

```
  (string key, string array[], int n) {
```

```
    return BinarySearch(key, array, 0, n - 1);
```

```
}
```



# Recursive Binary Search

```
template <typename T>
int BinarySearch
    (T key, T array[], int low, int high) {
    if (low > high) return -1;
    int mid = (low + high) / 2;
    if (key == array[mid]) return mid;
    if (key < array[mid])
        return BinarySearch(key, array, low, mid - 1);
    else
        return BinarySearch(key, array, mid + 1, high);
}
```

```
template< typename T>
int FindItemInSortedArray(T key, T array[], int n) {
    return BinarySearch(key, array, 0, n - 1);
}
```

# Template Functions

- A **template function** is a **generic function** that takes a **type parameter**.
- This parameter is defined implicitly by the types passed when function is called.
  - `string words[] ....`
  - `string wordToFind;`
  - `int location =  
    FindStringInSortedArray  
    (wordToFind, words, words.size());`

# BS in STL

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

bool myfunction (int i,int j) { return (i<j); }

int main () {
    int myints[] = {1,2,3,4,5,4,3,2,1};
    vector<int> v(myints,myints+9);

    // using default comparison:
    sort (v.begin(), v.end());

    cout << "looking for a 3... ";
    if (binary_search (v.begin(), v.end(), 3))
        cout << "found!\n"; else cout << "not found.\n";

    // using myfunction as comp:
    sort (v.begin(), v.end(), myfunction);

    cout << "looking for a 6... ";
    if (binary_search (v.begin(), v.end(), 6, myfunction))
        cout << "found!\n"; else cout << "not found.\n";

    return 0;
}
```



# Mutual Recursion

- Assume we want a function to find if a non-negative integer is even odd. I
- One inefficient but simple strategy is
  - A number is even if its predecessor is odd.
  - A number is odd if is not even.
  - The number 0 is even by definition.

# Mutual Recursion

```
bool IsEven(unsigned int n) {  
    if (n == 0)  
        return true;  
    else  
        return IsOdd(n - 1);  
}
```

```
bool IsOdd(unsigned int n) {  
    return !IsEven(n);  
}
```



# 5. Fibonacci Numbers

- Fibonacci Numbers:
  - $\text{Fib}(N) = \text{Fib}(N-1) + \text{Fib}(N-2)$
- They can be calculated using two different techniques
  - Recursion
  - Iteration



# Recursive Calculation of Fibonacci Numbers

- Recursive calculation of Fibonacci Numbers:

$$\text{Fib}(1) = 1$$

$$\text{Fib}(2) = 1$$

$$\text{Fib}(N) = \text{Fib}(N-1) + \text{Fib}(N-2)$$

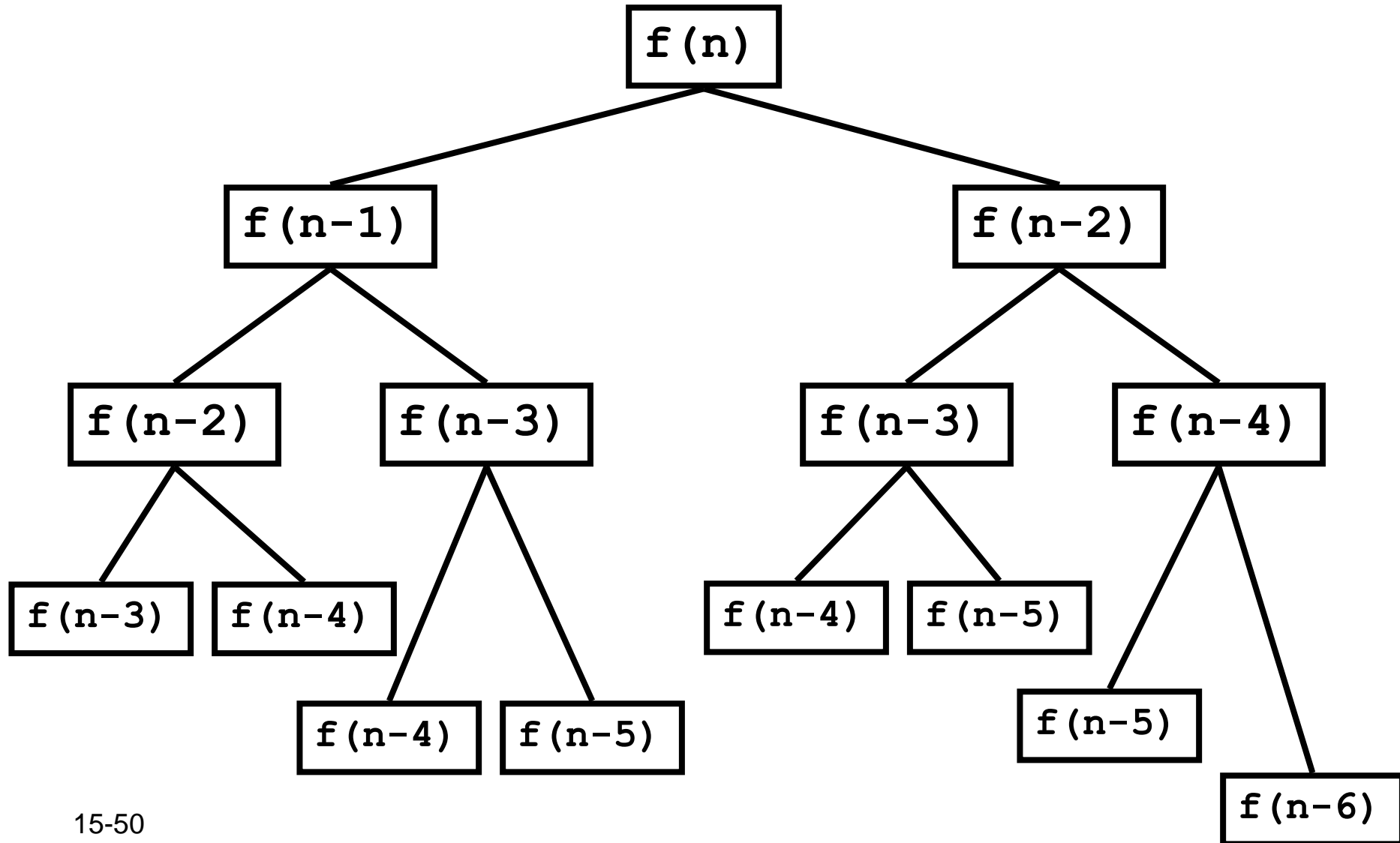
So:

$$\text{Fib}(3) = \text{Fib}(2) + \text{Fib}(1)$$

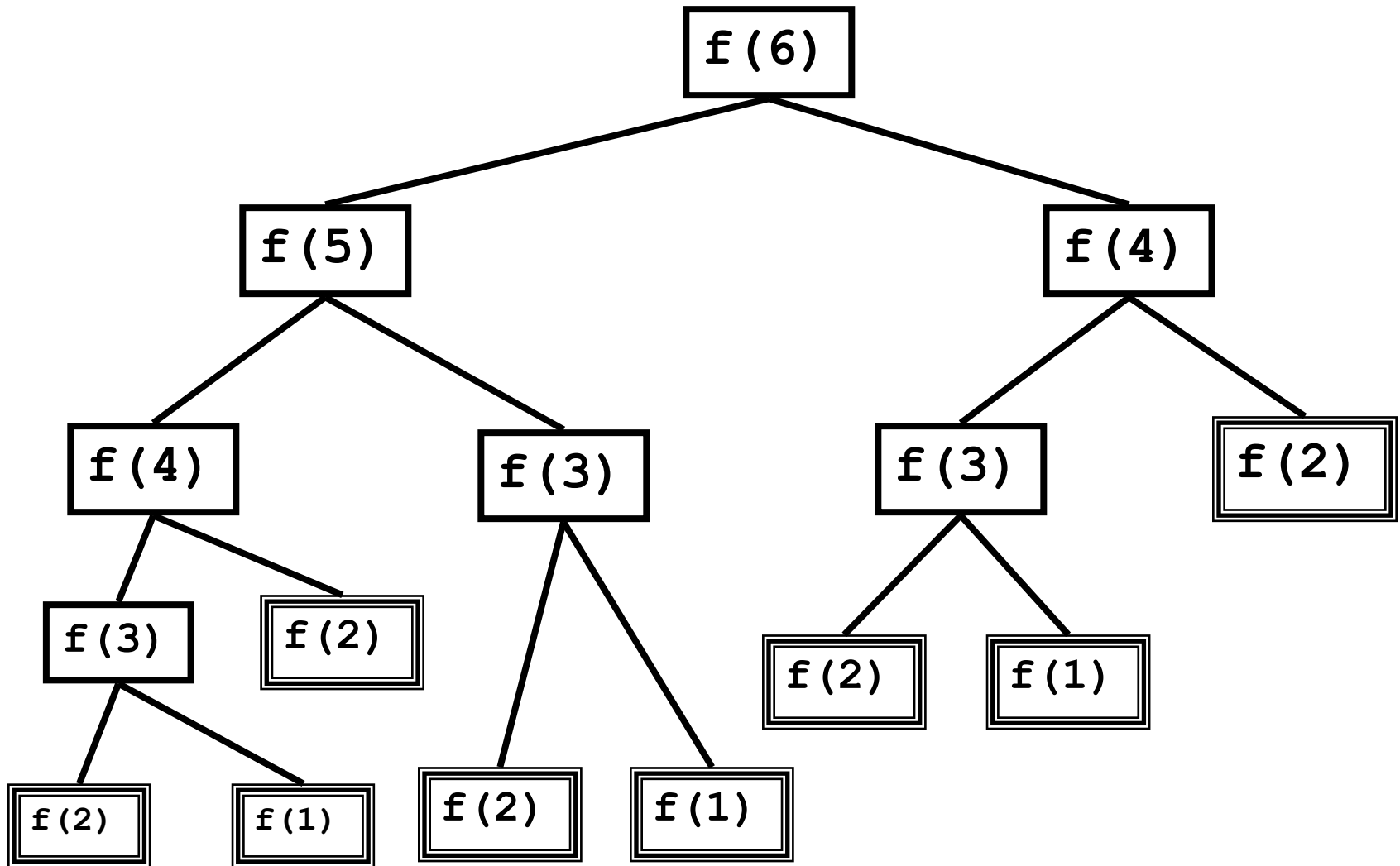
$$= 1 + 1$$

$$= 2$$

# Recursion Tree



# Recursion Tree



# Recursive Fib

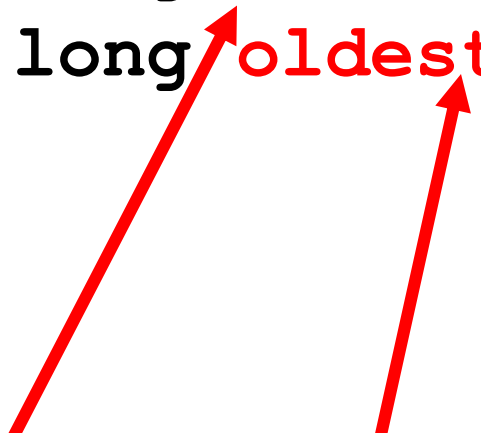
```
long Fib(long n) {  
    if (n < 2)  
        return n;  
    else  
        return Fib(n - 1) + Fib(n - 2);  
}
```

# Iterative Fib

```
long Fib(long n) {  
    long oldest = 1;  
    long old = 1;  
    long fib = 1;  
    while(n-- > 2) {  
        fib = old + oldest;  
        oldest = old;  
        old = fib;  
    }  
    return fib;  
}
```

# Better Recursive Fib

```
long long recFib (int n, long long old,  
                  long long oldest)  
{  
    if ( n < 1) return -1;  
    if (n == 1) return oldest;  
    if (n == 2) return old;  
    return recFib (--n, old+oldest, old);  
}
```



```
long long goodRecFib (int n){  
    return recFib(n, 1, 1);  
}
```