CS201: Data Structures and Discrete Mathematics I

Recursion

Recursive Definitions

- Recursive definition (or inductive definition): A definition in which the item being defined appears as part of the definition.
- Contain two parts:
  - A base, where some simple cases of the item being defined are given
  - An recursive step, where new cases of the item being defined are given in terms of previous cases.

Examples

- Fibonacci numbers

  $F(1) = 1, F(2) = 1$

  $F(n) = F(n-2) + F(n-1)$ for $n > 2$.

  (1, 1, 2, 3, 5, 8, 13, 21…)
- Recurrence relation: A rule like F(n), which define a sequence value in terms of one or more earlier values.
- Define n! recursively

  $1! = 1$

  $n! = n(n-1)!$ for $n > 1$

Recursively defined sequences

- A sequence S represents a list of objects that are enumerated in some order.
  - E.g, 1. $S(1) = 2$
    2. $S(n) = 2S(n-1)$ for $n \geq 2$
  - 2, 4, 8, 16, 32, …
- Another sequence T

1. $T(1) = 1$
2. $T(n) = T(n-1) + 3$ for $n \geq 2$

Recursively defined sets

- Define a set of people who are ancestors of James:
  1. James parents are ancestors of James.
  2. Every parent of an ancestor is an ancestor of James
- An identifier in a programming language can be alphanumeric strings of any length but must begin with a letter.
  1. A single letter is an identifier.
  2. If B is an identifier, so is the concatenation of B and any letter or digit.

Recursively defined operations

- A recursive definition of multiplication of two positive integers m and n is

    1. $m(1) = m$
    2. $m(n) = m(n-1) + m \quad$ for $n \geq 2$

- Let x be a string. Define the operation x^n (concatenation of x with itself n times) for $n \geq 1$

    1. $x^1 = x$
    2. $x^n = x^{n-1}x$ for $n \geq 1$

Recursive Programming:
Recursively defined algorithms

- Recursively computes the value of S(n)

```
S(integer n)
    If n = 1 then
        return 2
    else
        return 2*S(n-1)
    endif
end
```

Recursion programming - Basic Idea

- When writing recursive programs, we need
  - Base cases: we must always have some base cases, which can be solved without recursion.
  - Making progress: For the cases that are to be solved recursively, the recursive call must always make progress toward a base case.

Iteration versus Recursion

- Most of the time, we can express a problem more elegantly using recursion
- e.g. summation of numbers from 1 to n

```
sum(n)= n+(n-1)+(n-2)+...+2+1
```

$$= \sum_{i=1}^{n} i$$

```
→ sum(n)
      for (i=1,sum=0;i<=n;i++)
          sum=sum+i;
      return sum;
```

In Recursion

- Summation of numbers from 1 to n using *recursion*.

$$\texttt{sum(n)} = \texttt{n+(n-1)+(n-2)+...+2+1}$$

$$= \begin{cases} \texttt{1} & \texttt{if (n==1)} \\ \texttt{n+sum(n-1)} & \texttt{if (n>1)} \end{cases}$$

```
→ sum(n)
     if (n==1) return 1;
     else return n+sum(n-1);
```

Another Example of Recursion

- Product of numbers from 1 to n using recursion

```
fact(n) =   n*(n-1)*(n-2)*...*2*1

          = { 1              if (n==1)
            { n*fact(n-1)    if  (n>1)

-> fact(n)
      if (n==1) return 1;
      else return n*fact(n-1);
```

Visualizing Recursive Execution

- With nonrecursive programs, it is natural to visualize execution by imagining control stepping through the source code
  - This can be confusing for programs containing recursion
  - Instead, useful to imagine each call of a function generating a copy of the function, so that if the same function is called several times, several copies are present.

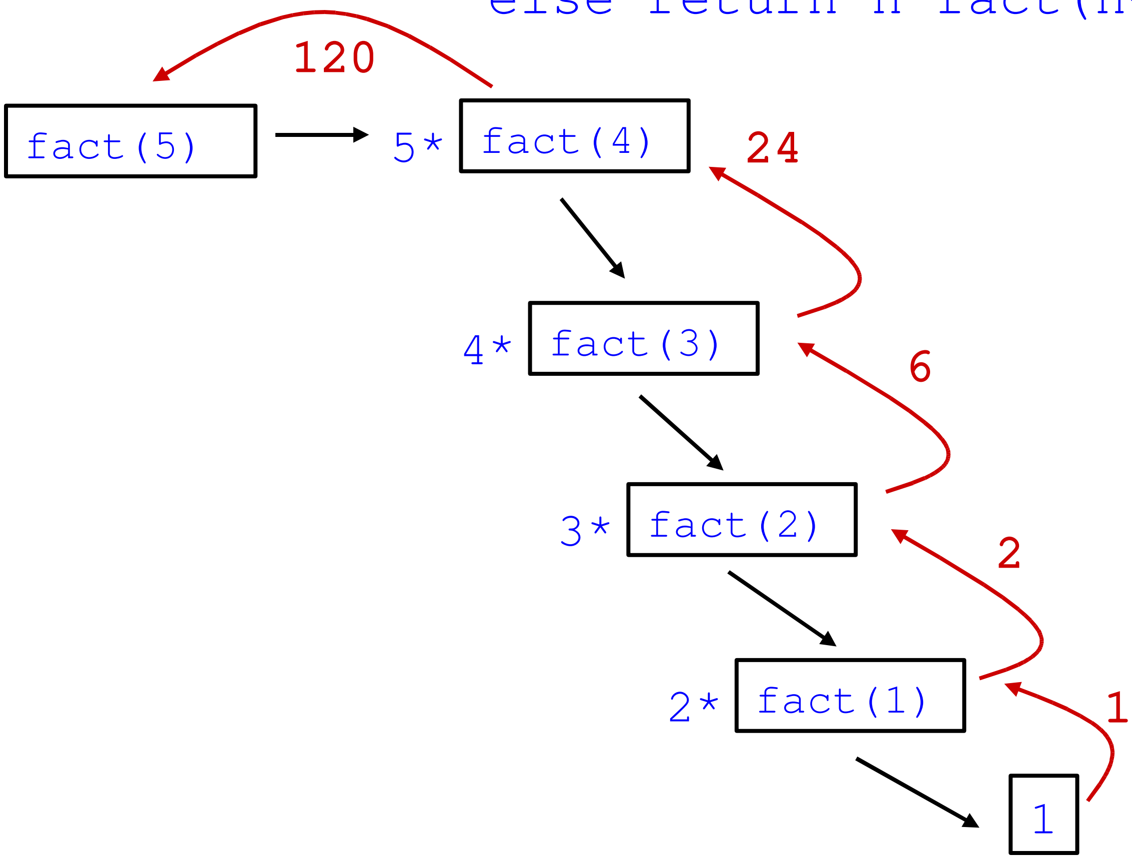
Scope

- When a function is called (stack takes role for the process)
  - caller is suspended
  - “state” of caller saved
  - new space allocated for variables of new function
  - …
  - end of new function
    - release the space allocated
    - return to the point next to the caller with the previous “state” recovered
- With recursive call, same things happen

How Recursion Works

- Given

```
fact(n) =   if (n==1) return 1;
            else return n*fact(n-1);
```



Computing x^n

- This is a simple program.

```
float power (float x, int n)
{
    if (!n)                    /* if (n==0) */
        return 1
    else
        return x * power(x, n-1)
}
```

What does this program do?

- This program is not easy to understand.

```
void f ()
 { int ch;
   if ((ch = getchar()) != '\n')
   {
      f();
      printchar(ch);
   }
 }
```

Given the input string "Is it going to work?"

Recursion - how to

Ask the following

- How can you solve the problem using the solution of a “simpler” instance of the problem?
- Can you be sure to have a “simplest” input? (If so, include separate treatment of this case.)
- Can you be sure to reach the “simplest” input?

Another Example: Merge Sort

- We now use another complex example to show the working of a recursive program.
- Sorting is the process of rearranging data in either ascending or descending order.
  - (2, 4, 1, 6, 5, 9, 2) => (1, 2, 2, 4, 5, 6, 9)
- We need sorting because
  - The data in sorted order is required
  - It is the initialization step of many algorithms.

Merge Sort: one sorting algorithm

- A nice example of a recursive algorithm.
- It is a divide-and-conquer algorithm
- Divide-and-conquer is an important technique in Computer Science. It solves problem in three steps:
  - Divide Step: divide the large problem into two or more smaller problems.
  - Recursively solve the smaller problems
  - Conquer Step: based on the results of the smaller problems, produce the result of the large problem.

Merge Sort Idea

- Divide Step: Divide the array into two equal halves
- Recursively sort the two halves
- Conquer Step: Merge the two halves to form a sorted array

An example

7	2	6	3	8	4	5

Divide into two equal halves

7	2	6	3

8	4	5

Recursively sort the halves

2	3	6	7

4	5	8

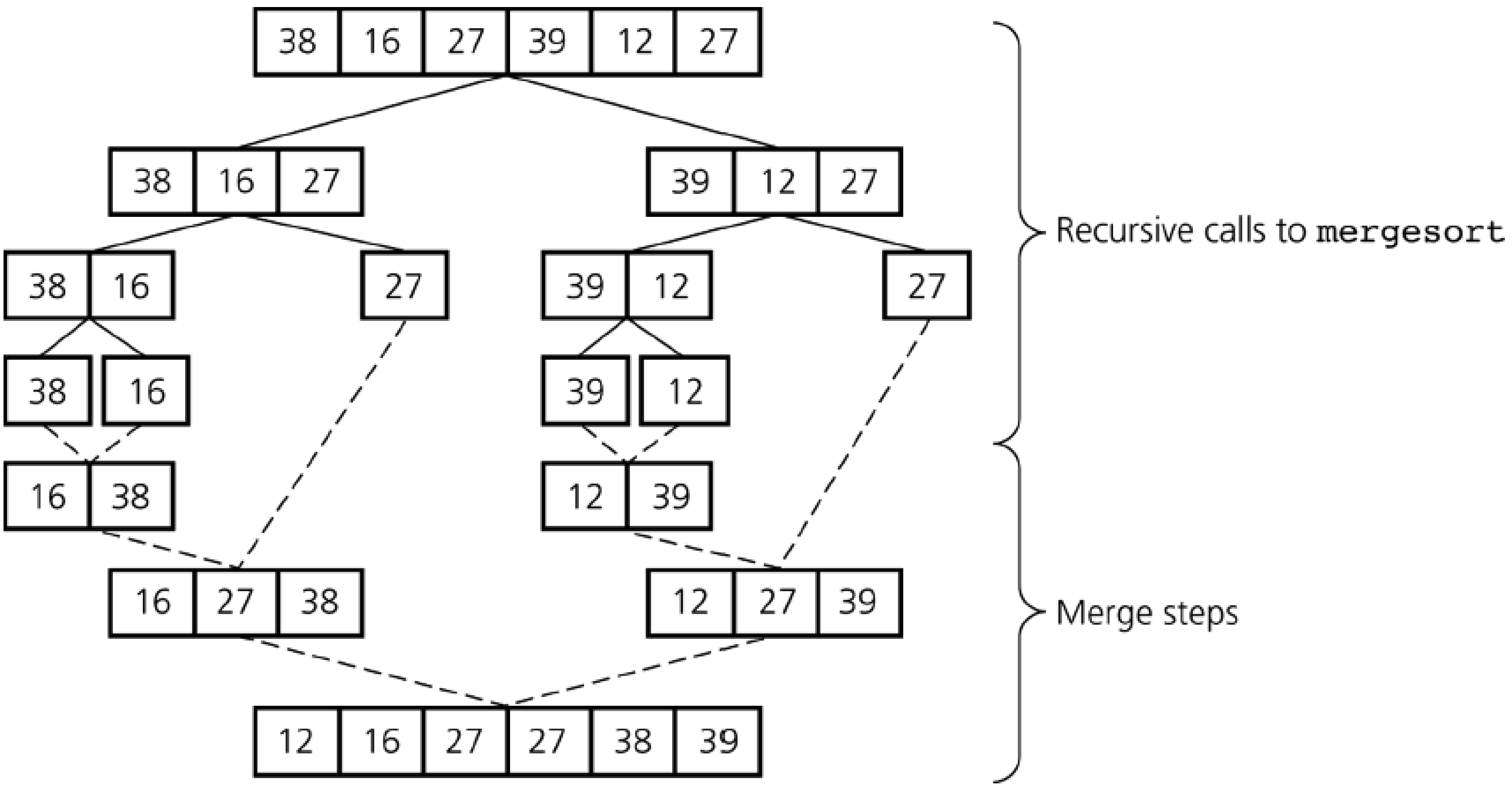
Merge them

2	3	4	5	6	7	8

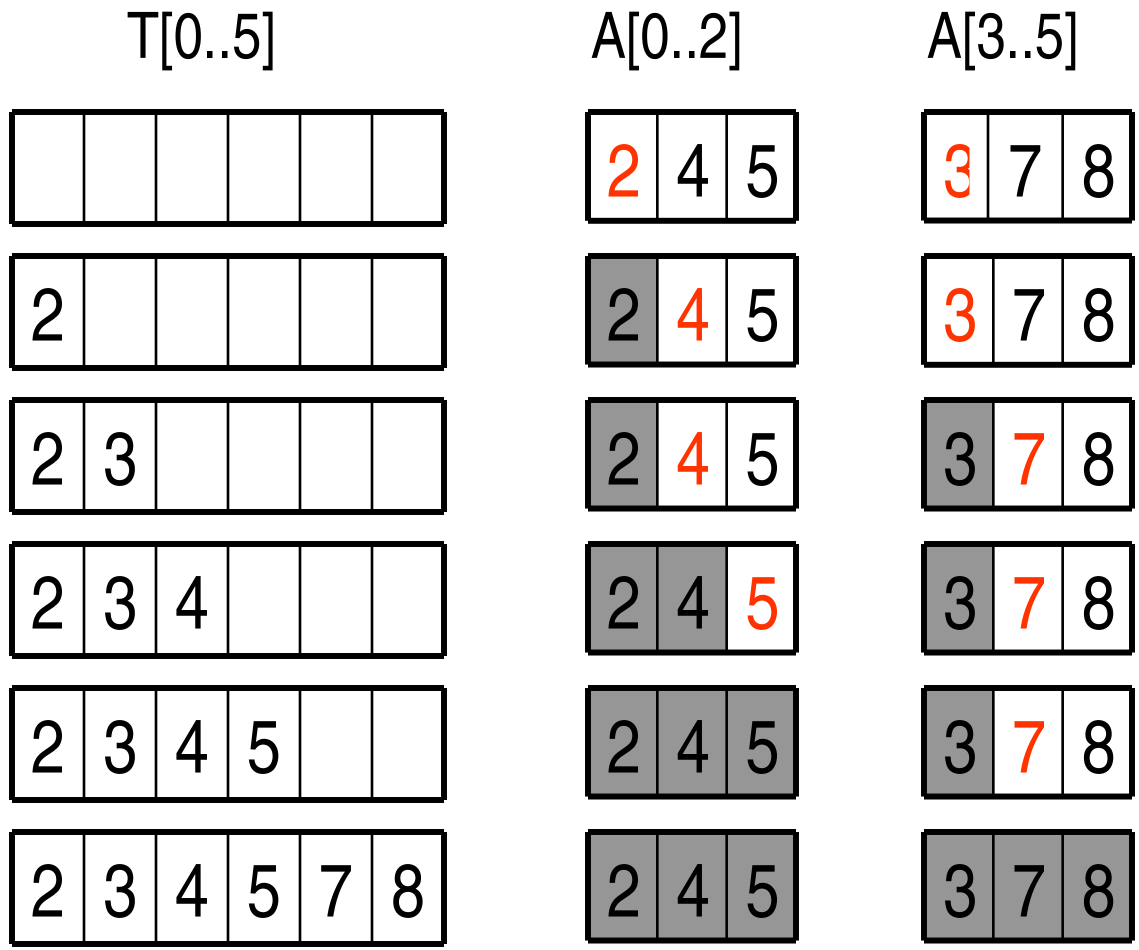
Merge Sort Algorithm

```
MergeSort(A[i..j])
if (i < j) {
    mid = (i+j)/2
    MergeSort(A[i..mid]);
    MergeSort(A[mid+1..j]);
    Merge(A[i..mid], A[mid+1..j]);
}
```

Merge Sort of an array of six integers



How to merge two subarrays?

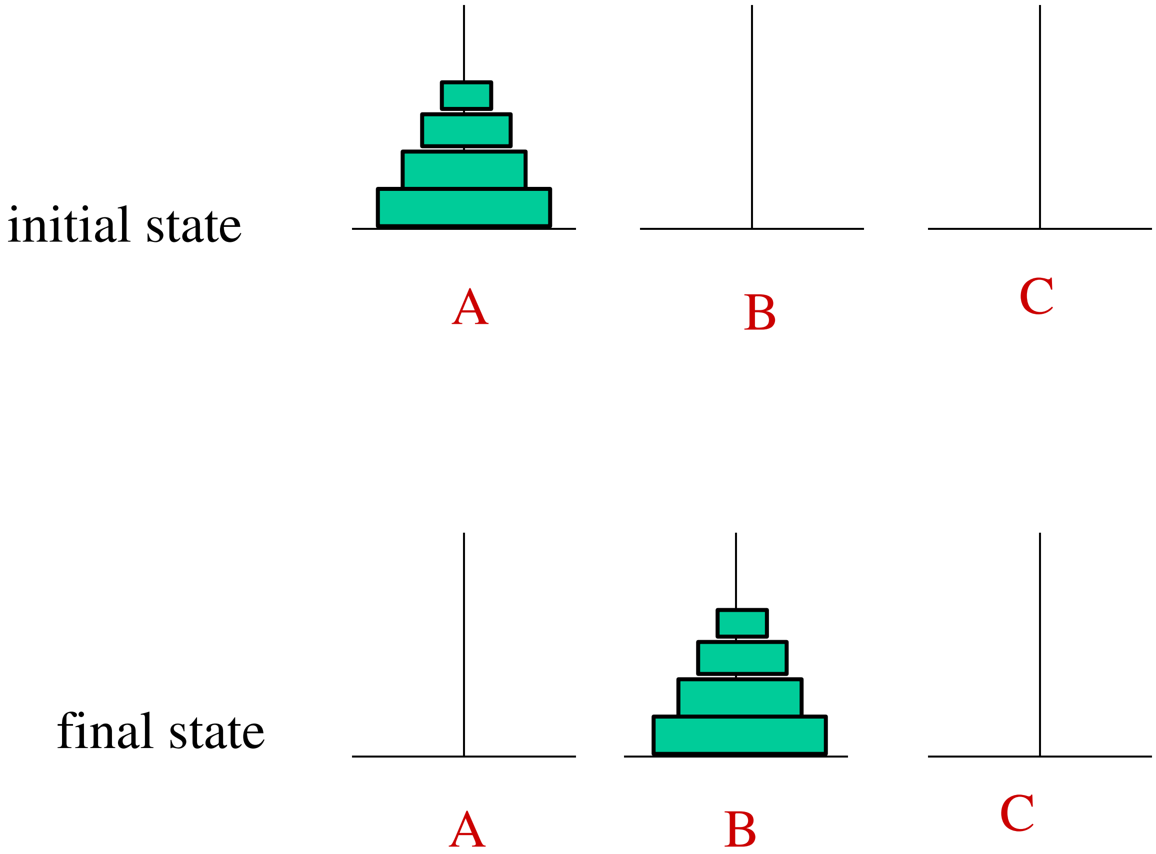


Merge Algorithm

Algorithm Merge(A[i..mid], A[mid+1..j])

2. While both subarrays are not empty,
   - Between the first entries of both subarrays, copy the smaller item into the first available entry in the temporary array T[].
3. When one subarray is empty,
   - finish off the nonempty subarray
4. Copy the result in T[] back to A[i..j]

Tower of Hanoi



Tower of Hanoi

```
void tower (int cnt, char A,char B,char C)
  {
    if (cnt==1)
        move(A,B);
    else {
        tower(cnt-1,A,C,B);
        move(A,B);
        tower(cnt-1,C,B,A);
         };
  }
```

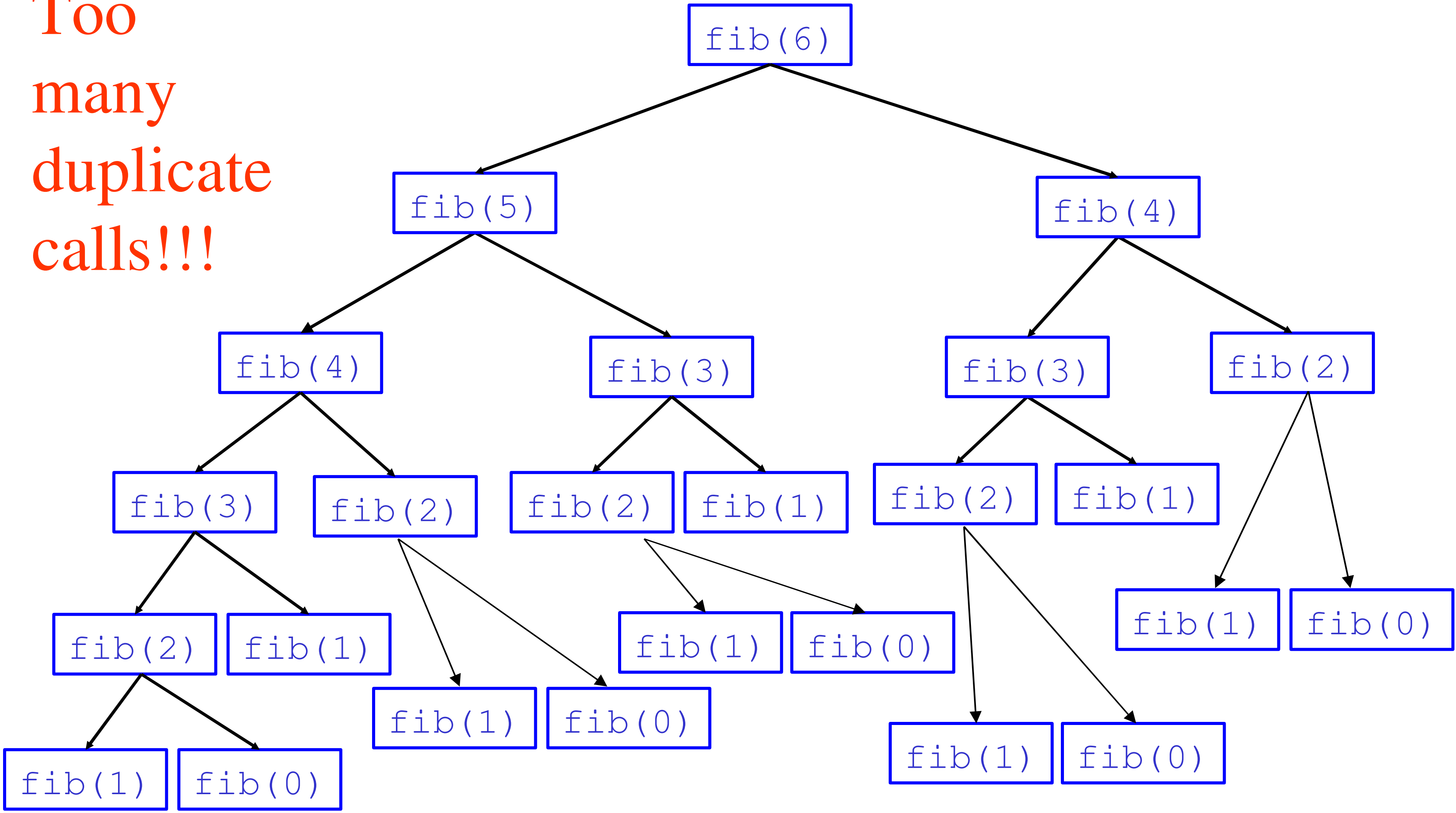
Fibonacci Numbers again

- Fibonacci numbers: 0,1,1,2,3,5,8,…
- First two are 0, and 1, rest are obtained by adding the previous two.
- Naïve method, using recursion:

```
int fib(int n)
{
    if (n < 2)
        return n;
        /* else */
        return fib(n-1)+fib(n-2);
}
```

Tracing Fibonacci Calls

Too many duplicate calls!!!



fib(int n) is extremely inefficient

n	Number of additions	Number of calls
6	12	25
10	88	177
15	986	1973
20	10945	21891
25	121392	242785
30	1346268	2692537

Much better to write an iterative function

```
int fib(int n)
{
   int fibn=0, fibn1=0, fibn2=1;

   if (n < 2)
      return n
   else
     {
        for( int i = 2; i <= n; i++ ) {
           fibn = fibn1 + fibn2;
           fibn1 = fibn2;
           fibn2 = fibn;
        }
     return fibn;
     }
}
```

Recursion or Iteration

- Every recursive procedure can be converted into an iterative version (sometime not a trivial task)
- No general rules prescribing when to use recursion and when not to.
- Recursion code is usually easily readable, simpler and clearer.
- The main problem with recursion is the hidden bookkeeping cost. Recursion is usually less efficient than its iterative equivalent.

Summary

- Inductive proof is perhaps the most commonly used proof technique in CS.
  - Base case
  - Inductive case.
- Recursion definition
  - A base
  - A recursive step
- An recursive program is often simpler and clearer, but can be less efficient than its iterative version.