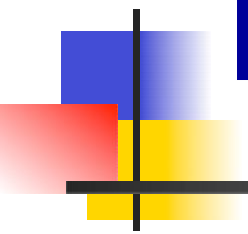


Lecture

Chapter 4. Induction and Recursion

Recursion:

All recursive algorithms must have the following:

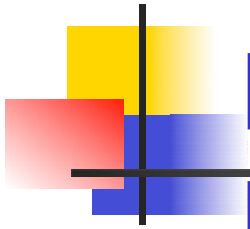
1. Base Case (i.e., when to stop)
2. Work toward Base Case
3. Recursive Call (i.e., call ourselves)

The "work toward base case" is where we simplify the problem (e.g., divide list into two parts, each smaller than the original). The recursive call, is where we use the same algorithm to solve a simpler version of the problem. The base case is the solution to the "simplest" possible problem (For example, the base case in the problem 'find the largest number in a list' would be if the list had only one number... and by definition if there is only one number, it is the largest).

For example, we can define the operation "find your way home" as:

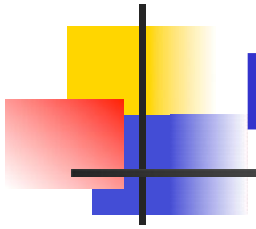
1. If you are at home, stop moving.///base case
2. Take one step toward home.///work towards base case
3. "find your way home".///recursive call..call ourself

Here the solution to finding your way home is two steps (three steps). First, we don't go home if we are already home. Secondly, we do a very simple action that makes our situation simpler to solve. Finally, we redo the entire algorithm.



Recursive Definitions

- In ***recursive definitions***, we *define* a function, a predicate, a set, or a more complex structure over an infinite domain (universe of discourse) by:
  - defining the function, predicate value, set membership, or structure of larger elements in terms of those of smaller ones.



Recursion

- ***Recursion*** is the general term for the practice of defining an object in terms of *itself*
  - or of part of itself.
  - This may seem circular, but it isn't necessarily.
- There are also recursive *algorithms*, *definitions*, *functions*, *sequences*, *sets*, and other structures.

Recursively Defined Function

We use two steps to define a function with the set of nonnegative integers as its domain:

BASIS STEP: Specify the value of the function at zero.

RECURSIVE STEP: Give a rule for finding its value at an integer from its values at smaller integers.

- Simplest case: One way to define a function $f: \mathbf{N} \rightarrow S$ (for any set S) or series $a_n = f(n)$ is to:
  - Define $f(0)$
  - For $n > 0$, define $f(n)$ in terms of $f(0),\ldots,f(n-1)$

- **Example**: Define the series $a_n = 2^n$ where n is a nonnegative integer recursively:
  - a_n looks like $2^0, 2^1, 2^2, 2^3,\ldots$
  - Let $a_0 = 1$ a1=2
  - For $n > 0$, let $a_n = 2 \cdot a_{n-1}$

Another Example

- Suppose we define $f(n)$ for all $n \in \mathbf{N}$ recursively by:
  - Let $f(0) = 3$//// **n=0**
  - For all $\boldsymbol{n} > \mathbf{0}$, let $f(n) = 2 \cdot f(n-1) + 3$

- What are the values of the following?
  - $f(1)$ = $2 \cdot f(0) + 3 = 2 \cdot 3 + 3 = 9$
  - $f(2)$ = $2 \cdot f(1) + 3 = 2 \cdot 9 + 3 = 21$
  - $f(3)$ = $2 \cdot f(2) + 3 = 2 \cdot 21 + 3 = 45$
  - $f(4)$ = $2 \cdot f(3) + 3 = 2 \cdot 45 + 3 = 93$

Recursive Definition Example: Factorial

- Give an inductive (recursive) definition of the factorial function,

$$F(n) = n! = \prod_{1 \le i \le n} i = 1 \cdot 2 \ldots n \qquad n \cdot F(n-1)$$

  - Basis step: $F(1) = 1$
  - Recursive step: $F(n) = n \cdot F(n-1)$ for $n > 1$
    - $F(2) = 2 \cdot F(1) = 2 \cdot 1 = 2$
    - $F(3) = 3 \cdot F(2) = 3 \cdot \{2 \cdot F(1)\} = 3 \cdot 2 \cdot 1 = 6$
    - $F(4) = 4 \cdot F(3) = 4 \cdot \{3 \cdot F(2)\} = 4 \cdot \{3 \cdot 2 \cdot F(1)\}$

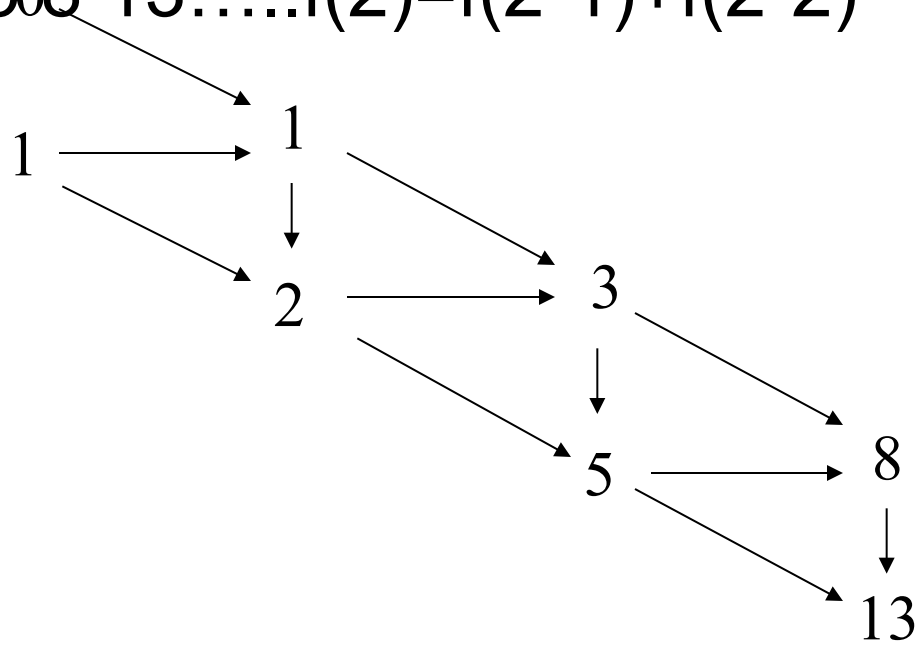
$$= 4 \cdot 3 \cdot 2 \cdot 1 = 24$$

The Fibonacci Numbers

- The ***Fibonacci numbers*** $f_{n\geq 0}$ is a famous series defined by:

$$f_0 = 0, \quad f_1 = 1, \quad f_{n\geq 2} = f_{n-1} + f_{n-2}$$

0 1 1 2 3 5 8 13…..f(2)=f(2-1)+f(2-2)



Recursively Defined Sets

- An infinite set S may be defined recursively, by giving:
  - A small finite set of *base* elements of S.
  - A rule for constructing new elements of S from previously-established elements.
- These:
  - Implicitly, no other elements of S exist.

base element (basis step)

construction rule (recursive step)

- **Example:** Let $3 \in S$, and let $x+y \in S$ if $x,y \in S$. What is S?

Example *cont.*

- Let $3 \in S$, and let $x+y \in S$ if $x, y \in S$. What is S?
  - $3 \in S$ (*basis step*) { 3,6,9,12,15,18,
  - 6 (= 3 + 3) is in S (*first application of recursive step*)
  - 9 (= 3 + 6) and 12 (= 6 + 6) are in S (*second application of the recursive step*)
  - 15 (= 3 + 12 or 6 + 9), 18 (= 6 + 12 or 9 + 9), 21(= 9 + 12), 24 (= 12 + 12) are in S (*third application of the recursive step*)
  - Therefore, so on **S = {3,** 6, 9, 12, 15, 18, 21, 24,…}
    = set of *all positive multiples of* 3

Recursive Algorithms

- Recursive definitions can be used to describe functions and sets as well as *algorithms*.
- A *recursive procedure* is a procedure that invokes itself.
- A *recursive algorithm* is an algorithm that contains a recursive procedure.
- *An algorithm is called recursive if it solves a problem by reducing it to an instance of the same problem with smaller input.*

Example

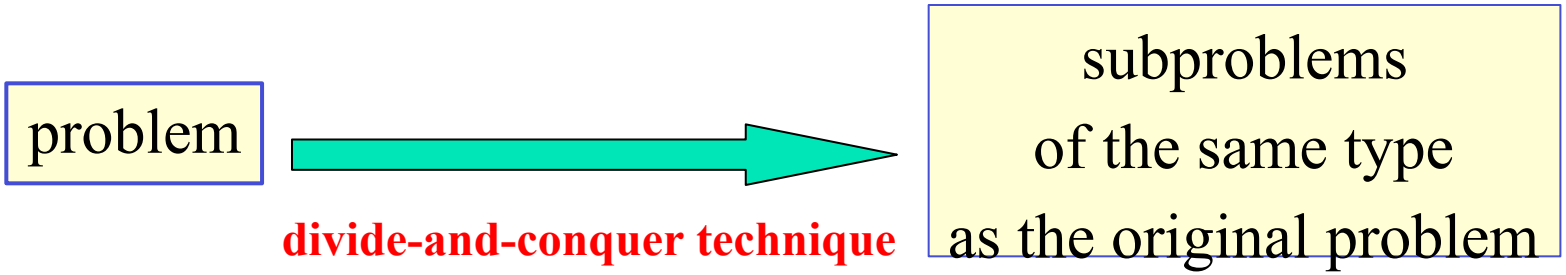
- A procedure to compute a^n.

$2._{0=}$ 1,$2._{1=}2$,$2._{2=}4$…2*

procedure *power*($a \neq 0$: real, $n \in \mathbf{N}$)

if $n = 0$ **then return** 1

else return $a \cdot power(a, n-1)$



Recursive Euclid's Algorithm

- gcd(a, b) = gcd((b mod a), a)

  procedure *gcd*(a,b∈**N** with $a < b$)
  if a = 0 **then return** b
  else return *gcd*(b **mod** a, a)

- Note recursive algorithms are often simpler to code than iterative ones…
- However, they can consume more stack space
  - if your compiler is not smart enough

Iterative code for gcd

```cpp
#include <iostream>


int gcd(int a, int b) {
    while (a != 0) {          // Continue until a becomes 0
        int temp = a;         // Store the current value of a
        a = b % a;            // Update a to b mod a
        b = temp;             // Update b to the old value of a
    }
    return b;                 // When a is 0, b contains the GCD
}
```

Recursive Linear Search

{Finds *x* in series *a* at a location ≥*i* and ≤*j*

procedure *search*

(*a*: series; *i*, *j*: integer; *x*: item to find)

if $a_i = x$ **return** *i* {At the right item? Return it!}

if *i* = *j* **return** 0 {No locations in range? Failure!}

return *search*(*a*, *i* +1, *j*, *x*) {Try rest of range}

- Note there is no real advantage to using recursion here over just looping

  for *loc* := *i* to *j*...

Iterative Fibonacci Algorithm

```
procedure iterativeFib(n ∈ N)  if n = 0 then
        return 0
    else begin
        x := 0
        y := 1
        for i := 1 to n – 1 begin
                z := x + y  x := y
                y := z
        end
    end
    return y   {the nth Fibonacci number}
```

Requires only $n - 1$ additions

Recursive Fibonacci Algorithm

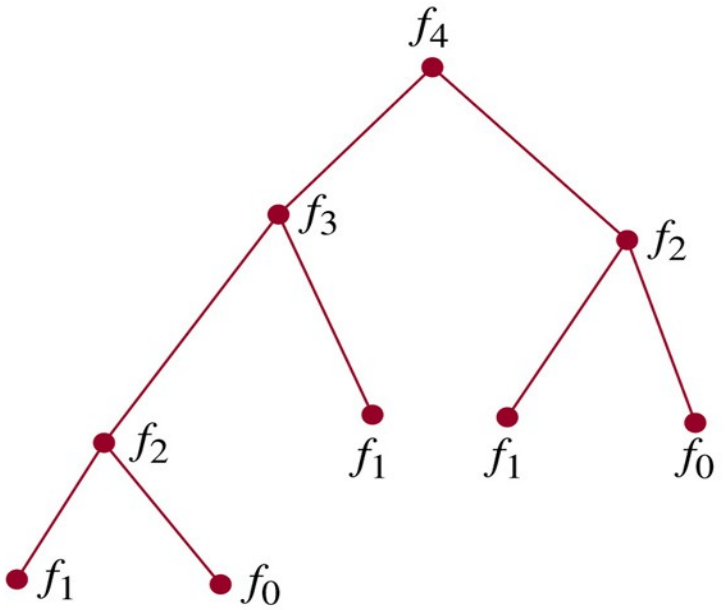
procedure *fibonacci*($n \in$ **N**)

if $n = 0$ **return** 0

if $n = 1$ **return** 1

return *fibonacci*($n - 1$) + *fibonacci*($n - 2$)

- How many additions are performed?



Recurrence relations

Definition: A recurrence relation expresses the nnn-th term of a sequence as a function of previous terms.

For example, the Fibonacci sequence is defined by the recurrence relation:
$F(n)=F(n-1)+F(n-2)$
with base cases $F(0)= 0$ and $F(1)=1$