What to do, and How to do it

A specification (or contract) tells us what some software must do; in its simplest form it states that

- if the input satisfies a given Precondition
- then the output must satisfy a given Postcondition

An algorithm is a description of how to implement a specification (fulfill a contract); it

- can be described in any standard programming language
- takes resources (time and space) that we would like to minimize.

Specifications written in English can often be ambiguous:

let y be the integer square root of integer x

and therefore it may be preferable to write them in logic.

Sorting and Selection

Why is sorting of interest?

- applications are ubiquitous
- there is a variety of algorithms
- some algorithms are "as good as we can get"

When is A' the result of sorting array A?

- when A' is non-decreasing (express in logic!)
- and a permutation of A

When is x the result of selecting the k'th smallest element of A?

- less that k elements are less than x
- at least k elements are less than or equal to x

Top-Down Approach to Algorithms

The Reduction principle:

> *Solve a "non-trivial" problem by using a solution to a "simpler" problem.*

A Simple Top-down Sorting

We may sort an array $A[1..n]$ as follows:

- If $n \leq 1$, then $A[1..n]$ is already sorted.

Otherwise, when $n > 1$, reduce larger instances of sorting to smaller instances:

1. sort $A[1..n-1]$; then
2. insert $A[n]$ into $A[1..n-1]$ at the proper location.

This is the insertion sort algorithm.

Input: $A[1..n]$ is an array of numbers.
Output: $A[1..n]$ is a permutation of its original values such that $A[1..n]$ is non-decreasing.

```
InsertionSort(A[1..n])
    if n > 1
        InsertionSort(A[1..n − 1])
        InsertLast(A[1..n])
```

When running it, the stack grows!

Bottom-up Computation

We can often save stack space by implementing a top-down design in a bottom-up fashion:

1. Compute solutions to the smallest instances.
2. Using the top-down solution as a guide, combine the solutions of smaller instances to obtain solutions to larger instances.

Example: the fibonacci function:

$$\begin{aligned} fib(0) = fib(1) &= 1 \\ fib(n+2) &= fib(n) + fib(n+1) \end{aligned}$$

- this takes exponential time if executed naively.
- but there is an efficient iterative program:

$\text{FIB\_ITER}(n)$

$\quad i, j \leftarrow 1$

$\quad$ **for** $k \leftarrow 1$ **to** $n - 1$

$\qquad i, j \leftarrow j, i + j$

$\quad$ **return** j

Insertion Sort, Bottom-Up

The top-down version

```
InsertionSort(A[1..n])
    if n > 1
        InsertionSort(A[1..n − 1])
        InsertLast(A[1..n])
```

unfolds to

```
InsertLast(A[1..2])
...
InsertLast(A[1..n])
```

and thus a bottom-up version is

```
InsertionSort(A[1..n])
    for i ← 2 to n
        InsertLast(A[1..i])
```

Insert the Last Element in Proper Place

Input: $A[1..n]$ with $A[1..n-1]$ non-decreasing
Output: $A[1..n]$ is a permutation of its original values such that $A[1..n]$ is non-decreasing

$\textsc{InsertLast}(A[1..n])$
if $n > 1$ **and** $A[n] < A[n-1]$
  $A[n] \leftrightarrow A[n-1]$
  $\textsc{InsertLast}(A[1..n-1])$

This is recursive but is

- good recursion: tail-recursion
- which easily converts to iteration:

$\textsc{InsertLast}(A[1..n])$
$j \leftarrow n$
while $j > 1$ **and** $A[j] < A[j-1]$
  $A[j] \leftrightarrow A[j-1]$
  $j \leftarrow j-1$

Iterative Insertion Sort

$\textsc{InsertionSort}(A[1..n])$
  for $i \leftarrow 2$ **to** n
    $j \leftarrow i$
    while $j > 1$ **and** $A[j] < A[j-1]$
      $A[j] \leftrightarrow A[j-1]$; $j \leftarrow j - 1$

Space use: only a constant (strictly speaking logarithmic in n) amount of extra space is needed. We say the algorithm is in-place.

Time use is:

- best-case: linear, i.e., proportional to n
- worst-case: quadratic, i.e., proportional to n^2
- average case: ???
  (depends on expected input distribution)