CIS 575. Introduction to Algorithm Analysis
Material for January 19, 2024

Insertion Sort: Recursive Implementation

1 Recursive Insertion Sort

Recall that we expressed the *insertion sort* algorithm by the recursive function

```
**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($A[1..n]$)
```

We must demand that InsertLast satisfies the specification (to be implemented later):

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

1.1 Problem: Stack Use

For each recursive call to InsertionSort we need to store on the stack

- the address in the code to return to
- the value of n as perceived by the caller (the callee will perceive it to be one smaller).

Essentially, the stack must contain enough information to carry out all the postponed calls to InsertLast. Thus the stack will grow at a rate proportional to n; this may not seem like a big deal, but can still be quite bad as stack space is often severely limited.

1.2 Remedy: Bottom-up

We shall therefore look for another implementation, based on the *bottom-up* principle. In general, for an algorithm it is often the case that

- it is most conveniently *designed* using the top-down approach
- its *efficient implementation* requires a bottom-up approach.

2 Bottom-Up Computation

The *bottom-up* approach may be described as follows:

1. first compute solutions to the smallest instances
2. next, using the top-down solution as a guide, combine the solutions of smaller instances into solutions to larger instances.

A well-known example, showing the potential immense power of the bottom-up approach, is the *fibonacci* function which may be defined by the recursive equations

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

which are straightforward to translate into a recursive function, but the naive execution of that function will take *exponential* time. On the other hand, a bottom-up approach may yield a much more efficient *iterative* program:

```
FIB_ITER($n$)
  $i, j \leftarrow 1, 1$
  for $k \leftarrow 1$ to $n - 1$
    $i, j \leftarrow j, i + j$
  return $j$
```

where we (following Rodney Howell's textbook) use a somewhat non-standard notation for assignments:

Assignment Notation We shall write $x \leftarrow E$ for the assignment of the value of expression E to the variable x. We believe that this is to be preferred to the much more common notation $x = E$ which may convey the false impression that assignment is a symmetric concept (alternatively, we could write $x := E$ as in say Pascal).

We also allow parallel assignments: $x, y \leftarrow E_1, E_2$ means that

1. the expressions E_1 and E_2 (which may contain x and/or y) are evaluated to values v_1 and v_2, and next
2. v_1 is assigned to x, and v_2 is assigned to y.