Foundations of Algorithms
Homework 0

Arthur Nunes-Harwitt

Python programmers: It is permissible to use Python lists; you can use indexing to access the first element, you can use slicing to compute the tail; you can compare to the empty list — do *not* check if the length of the list is zero; and you can use + to put an element at the beginning. Of course, you should give names to all these operations.

Java programmers: Translating list operations into Java is more challenging. It is preferable to write your own classes to implement lists. Java collections emphasize destructive operations, which you should *not* use. However, it is possible to use the `LinkedList` class as follows. You can use `getFirst` to access the first element; you can use `subList` to compute the tail; you can use `isEmpty` to check for the empty list; and you can use `addFirst` to add an element to the beginning *as long as you* `clone` *the list first*.

1. Calculate $\mathrm{iSort}([4, 1, 3, 2])$ as in the notes showing every step.

2. Look up the selection-sort algorithm. Translate the algorithm into functional pseudo-code. (Note that selection does not require swapping. You may find it helpful to test your code using ALTO.)

3. **(project)** Translate the following pseudo-code into working code. The function should be named `r`.

$$\begin{array}{lcl} r([]) & = & [] \\ r(x :: xs) & = & r(xs) + [x] \end{array}$$

4. **(project)** Translate the following pseudo-code into working code. The function should be named `prod`.

$$\begin{array}{lcl} 0 \odot n & = & 0 \\ (m+1) \odot n & = & (m \odot n) + n \end{array}$$

5. **(project)** Translate the following pseudo-code into working code. The function should be named `fastPow`. (Note that b^2 does *not* involve a recursive call; it is just squaring.)

$$\begin{array}{lcl} b^0 & = & 1 \\ b^{2k} & = & (b^2)^k \\ b^{2k+1} & = & (b^2)^k \times b \end{array}$$

6. **(project)** Translate the following pseudo-code into working code. The function should be named `prodAccum`.

$$\begin{array}{lcl} \mathrm{prodAccum}(0, n; a) & = & a \\ \mathrm{prodAccum}(m+1, n; a) & = & \mathrm{prodAccum}(m, n; n+a) \end{array}$$

7. **(project)** Translate the following pseudo-code into working code. The function should be named `minChange`. You will also need to write code for min and $\oplus$. (See page 16 in the notes for explanations of min and $\oplus$.)

$$\begin{array}{lcl} \mathrm{minChange}(0, ds) & = & 0 \\ \mathrm{minChange}(a, []) & = & \mathsf{Failure} \\ \mathrm{minChange}(a, d :: ds) & = & \begin{cases} \mathrm{minChange}(a, ds) & \text{if } d > a \\ \min(1 \oplus \mathrm{minChange}(a-d, d :: ds), \mathrm{minChange}(a, ds)) & \text{otherwise} \end{cases} \end{array}$$

8. (**project**) Translate the following pseudo-code into working code. The function should be named `greedyMinChange`.

$$
\begin{array}{lcl}
\mathrm{greedyMinChange}(0, ds) & = & 0 \\
\mathrm{greedyMinChange}(a, []) & = & \mathsf{Failure} \\
\mathrm{greedyMinChange}(a, d :: ds) & = & \left\{ \begin{array}{ll} \mathrm{greedyMinChange}(a, ds) & \text{if } d > a \\ q \oplus \mathrm{greedyMinChange}(r, ds) & \text{otherwise} \end{array} \right. \\
 & & \text{where } q = \text{quotient}(a, d), r = \text{remainder}(a, d)
\end{array}
$$

9. Consider the following pseudo-code.

$$
\begin{array}{lcl}
b^0 & = & 1 \\
b^{n+1} & = & b^n \times b
\end{array}
$$

(a) Transform the pseudo-code by adding an accumulation parameter and making it tail-recursive. It should continue to have the form of functional pseudo-code.

(b) Transform the tail-recursive functional pseudo-code into imperative pseudo-code. Then transform this imperative pseudo-code into iterative pseudo-code that has no recursive calls.

(c) (**project**) Translate the iterative imperative pseudo-code into working code. The function should be called `powIt`; it should take two arguments — the base and the exponent in that order.