Algorithms. To avoid thinking too much about non-mathematical programming issues, we will specify each algorithm we consider as a function. That is, each of our algorithms is a list of instructions that, given an input of some specified type or types, goes through the instructions, eventually (we hope) producing an output of some specified type. Note that if an algorithm always deterministically produces an output of the specified type, then it really is a function in the previous sense.

With that in mind, the rest of these notes describe several recursive algorithms, that is, functions whose value depends on running the same function on a “smaller” input. Induction is the most natural way to prove that such a function returns the result that we want, and so we see that:

Recursive programming is monetized induction.

Fast multiplication. Consider the function \texttt{mult}(b, n) defined (recursively) as follows. (The idea is to build up the operation of multiplication in terms of addition.)

- **Input:** A real number \(b\) and a positive integer \(n\).
- **Procedure:**
  1. If \(n = 1\), output the value \(\texttt{mult}(b, n) = b\).
  2. If \(n\) is odd, output the value
     \[
     \texttt{mult}(b, n) = b + \texttt{mult}(b + b, \frac{n - 1}{2}).
     \]
  3. Otherwise, output the value \(\texttt{mult}(b, n) = \texttt{mult}(b + b, n/2)\).

Note: One might object that we are computing multiplication using the more complicated operation of division, in that we need to compute \((n - 1)/2\) or \(n/2\). However, if \(n\) is written in terms of its binary digits, replacing \(n\) by \((n - 1)/2\) or \(n/2\) is just cutting off the last digit of \(n\), and so this algorithm again becomes practical.

Fast exponentiation. Consider the function \texttt{power}(b, n) defined (recursively) as follows.

- **Input:** A real number \(b\) and a positive integer \(n\).
- **Procedure:**
  1. If \(n = 1\), output the value \(\texttt{power}(b, n) = b\).
  2. If \(n\) is odd, output the value
     \[
     \texttt{power}(b, n) = b \ast \texttt{power}(b^2, \frac{n - 1}{2}),
     \]
     where \(\ast\) is multiplication.
3. Otherwise, output the value $\text{power}(b, n) = \text{power}(b^2, n/2)$.

- **Sample run:** To calculate $\text{power}(b, 13)$ for some real number $b$, we have:

$$
\begin{align*}
\text{power}(b, 13) &= b \times \text{power} \left( b^2, \frac{12}{2} \right) \\
&= b \times \text{power} \left( b^2, 6 \right) \\
&= b \times \text{power} \left( (b^2)^2, 3 \right) \\
&= b \times \text{power} \left( b^4, 3 \right) \\
&= b \times \left( b^4 \times \text{power} \left( (b^4)^2, \frac{2}{2} \right) \right) \\
&= b \times \left( b^4 \times \text{power} \left( b^8, 1 \right) \right) \\
&= b \times b^4 \times b^8 = b^{13}.
\end{align*}
$$

**Lists and sorting.** For us, a *list* will be a one-dimensional array of variable length. For example, a list of integers will be a finite sequence of integers $[a_1, \ldots, a_n]$. To discuss algorithms for sorting lists, we assume that the following features of our “language” already exist and work correctly. Suppose $\text{foo} = [a_1, \ldots, a_n]$ and $\text{bar} = [b_1, \ldots, b_m]$ are lists.

- $\text{foo}[k]$ outputs the $k$th entry in $\text{foo}$.

- $\text{length(}\text{foo})$ outputs the number of entries in $\text{foo}$, i.e., $n$. Note that the empty list $[]$ has length 0.

- $\text{head(}\text{foo}, k)$ outputs $[a_1, \ldots, a_k]$, e.g., $\text{head(}\text{foo}, 1)$ outputs a list containing only the first entry of $\text{foo}$.

- $\text{tail(}\text{foo}, k)$ outputs $[a_k, \ldots, a_n]$, e.g., $\text{tail(}\text{foo}, 2)$ outputs $\text{foo}$ with the first entry removed. We also set the convention that if $k > \text{length(}\text{foo})$, then $\text{tail(}\text{foo}, k)$ outputs the empty list $[]$.

- $\text{concat(}\text{foo}, \text{bar})$ outputs $[a_1, \ldots, a_n, b_1, \ldots, b_m]$ (the *concatenation* of $\text{foo}$ and $\text{bar}$).

- If $c$ is an object, $\text{append(}\text{foo}, c)$ outputs the list $[a_1, \ldots, a_n, c]$ (i.e., $\text{foo}$ with the element $c$ added to the end).

In the following algorithms, assume that all lists have entries that are ordered somehow, i.e., given list entries $a$ and $b$, either $a < b$, $a = b$, or $a > b$, in some appropriate sense.

**Merge two sorted lists.** Consider the function $\text{mergetwosortedlists(}\text{foo}, \text{bar})$, defined as follows:

- **Input:** Two lists $\text{foo}$, $\text{bar}$, which we assume are already sorted, i.e., $\text{foo}[1] \leq \text{foo}[2] \leq \ldots$ and $\text{bar}[1] \leq \text{bar}[2] \leq \ldots$.

- **Procedure:**

  1. If $\text{length(}\text{foo}) = 0$, output $\text{bar}$. 
2. If length(bar) = 0, output foo.
3. If foo[1] ≤ bar[1], then output the list
   concat(head(foo, 1), mergetwosortedlists(tail(foo, 2), bar)).
4. Otherwise, output the list
   concat(head(bar, 1), mergetwosortedlists(foo, tail(bar, 2))).

- **Sample run:** To calculate mergetwosortedlists([-1, 2, 3], [2, 5]), we have:

\[
\begin{align*}
\text{mergetwosortedlists} & \left(\left[-1, 2, 3\right], \left[2, 5\right]\right) \\
& = \text{concat}([-1], \text{mergetwosortedlists}([2], [2, 5])) \\
& = \text{concat}([-1], \text{concat}([2], \text{mergetwosortedlists}([3], [2, 5]))) \\
& = \text{concat}([-1], \text{concat}([2], \text{concat}([2], \text{mergetwosortedlists}([], [5])))) \\
& = \text{concat}([-1], \text{concat}([2], \text{concat}([2], \text{concat}([3], [5])))) \\
& = [-1, 2, 2, 3, 5].
\end{align*}
\]

**Merge a list of sorted lists.** Consider the function mergesortedlists(foolist), defined as follows:

- **Input:** A list foolish, each of whose entries is a sorted list.
- **Procedure:**
  1. If length(foolist) = 1, then output foolish[1].
  2. Otherwise, output the list

\[
\begin{align*}
\text{mergesortedlists} & \left(\text{append}(\text{tail}(\text{foolist}, 3), \text{mergetwosortedlists}(\text{foolist}[1], \text{foolist}[2])))\right).
\end{align*}
\]

- **Sample run:** Assuming mergetwosortedlists works as advertised, to calculate mergesortedlists([[[-1, 2, 3], [2, 5]], [1, 5, 6, 7], [-4, 0]]), we have:

\[
\begin{align*}
\text{mergesortedlists} & \left(\left[\left[-1, 2, 3\right], \left[2, 5\right]\right], \left[1, 5, 6, 7\right], \left[-4, 0\right]\right) \\
& = \text{mergesortedlists}\left(\text{append}(\left[\left[1, 5, 6, 7\right], \left[-4, 0\right]\right], \left[-1, 2, 2, 3, 5\right])\right) \\
& = \text{mergesortedlists}\left(\left[\left[1, 5, 6, 7\right], \left[-4, 0\right]\right], \left[-1, 2, 2, 3, 5\right]\right) \\
& = \text{mergesortedlists}\left(\text{append}(\left[\left[-1, 2, 2, 3, 5\right], \left[-4, 0, 1, 5, 6, 7\right]\right), \left[-4, 0, 1, 5, 6, 7\right]\right) \\
& = \text{mergesortedlists}\left(\left[\left[-4, 0, 1, 2, 2, 3, 5, 5, 5, 6, 7\right]\right]\right) \\
& = [-4, -1, 0, 1, 2, 2, 3, 5, 5, 6, 7].
\end{align*}
\]

**Merge sort.** Consider the function mergesort(foo), defined as follows:

- **Input:** A list foo.
- **Procedure:**
1. Create a list $\text{bar}$ such that $\text{bar}[k] = [\text{foo}[k]]$, i.e., the $k$th entry of $\text{bar}$ is a list of length 1 containing the $k$th entry of $\text{foo}$.

2. Output the list $\text{mergesortedlists}(\text{bar})$.

Problems

1. Prove that if $n$ is an integer and $n \geq 1$, then $\text{mult}(b, n) = nb$. Use induction on $n$.

2. Prove that if $n$ is an integer and $n \geq 1$, then $\text{power}(b, n) = b^n$. Use induction on $n$.

3. Prove that if $\text{foo}$ and $\text{bar}$ are lists that are already sorted, e.g., $\text{foo}[1] \leq \text{foo}[2] \leq \ldots$ and $\text{bar}[1] \leq \text{bar}[2] \leq \ldots$, then $\text{mergetwosortedlists}(\text{foo}, \text{bar})$ is a sorted list containing all of the entries of $\text{foo}$ and $\text{bar}$. (I.e., the answer is something like a sorted union, except an element is allowed to appear multiple times.) Use induction on $n = \text{length(\text{foo})} + \text{length(\text{bar})}$.

4. (a) Suppose $\text{foolist}$ is a list of lists, and each entry of $\text{foolist}$ is sorted. Prove by induction on $n = \text{length(\text{foolist})}$ that $\text{mergesortedlists}(\text{foolist})$ outputs a sorted list containing all of the entries of all of the elements of $\text{foolist}$.

(b) Prove that if $\text{foo}$ is a list, then $\text{mergesort}(\text{foo})$ outputs a list containing all of the entries of $\text{foo}$, but sorted. (There is no need for induction here; just apply part (a).)