Math 42
Algorithms, recursion, and induction

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
  $$\texttt{mult}(b, n) = b + \texttt{mult}\left(b + b, \frac{n-1}{2}\right).$$
  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
  $$\texttt{power}(b, n) = b * \texttt{power}\left(b^2, \frac{n-1}{2}\right),$$
  where $*$ is multiplication.

3. Otherwise, output the value $\texttt{power}(b, n) = \texttt{power}(b^2, n/2)$.

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

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

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 $\texttt{foo} = [a_1, \ldots, a_n]$ and $\texttt{bar} = [b_1, \ldots, b_m]$ are lists.

- $\texttt{foo}[k]$ outputs the kth entry in `foo`.
- $\texttt{length}(\texttt{foo})$ outputs the number of entries in `foo`, i.e., n. Note that the empty list $[\,]$ has length 0.
- $\texttt{head}(\texttt{foo}, k)$ outputs $[a_1, \ldots, a_k]$, e.g., $\texttt{head}(\texttt{foo}, 1)$ outputs a list containing only the first entry of `foo`.
- $\texttt{tail}(\texttt{foo}, k)$ outputs $[a_k, \ldots, a_n]$, e.g., $\texttt{tail}(\texttt{foo}, 2)$ outputs `foo` with the first entry removed. We also set the convention that if $k > \texttt{length}(\texttt{foo})$, then $\texttt{tail}(\texttt{foo}, k)$ outputs the empty list $[\,]$.
- $\texttt{concat}(\texttt{foo}, \texttt{bar})$ outputs $[a_1, \ldots, a_n, b_1, \ldots, b_m]$ (the *concatenation* of `foo` and `bar`).
- If c is an object, $\texttt{append}(\texttt{foo}, c)$ outputs the list $[a_1, \ldots, a_n, c]$ (i.e., `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 $\texttt{mergetwosortedlists}(\texttt{foo}, \texttt{bar})$, defined as follows:

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

1. If $\texttt{length}(\texttt{foo}) = 0$, output `bar`.

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

- **Sample run:** To calculate $\texttt{mergetwosortedlists}([-1, 2, 3], [2, 5])$, we have:

$$
\begin{aligned}
&\texttt{mergetwosortedlists}([-1, 2, 3], [2, 5]) \\
&= \texttt{concat}([-1], \texttt{mergetwosortedlists}([2, 3], [2, 5])) \\
&= \texttt{concat}([-1], \texttt{concat}([2], \texttt{mergetwosortedlists}([3], [2, 5]))) \\
&= \texttt{concat}([-1], \texttt{concat}([2], \texttt{concat}([2], \texttt{mergetwosortedlists}([3], [5])))) \\
&= \texttt{concat}([-1], \texttt{concat}([2], \texttt{concat}([2], \texttt{concat}([3], \texttt{mergetwosortedlists}([\ ], [5]))))) \\
&= \texttt{concat}([-1], \texttt{concat}([2], \texttt{concat}([2], \texttt{concat}([3], [5])))) \\
&= [-1, 2, 2, 3, 5].
\end{aligned}
$$

Merge a list of sorted lists. Consider the function $\texttt{mergesortedlists}(\texttt{foolist})$, defined as follows:

- **Input:** A list `foolist`, each of whose entries is a sorted list.
- **Procedure:**

1. If $\texttt{length}(\texttt{foolist}) = 1$, then output $\texttt{foolist}[1]$.
2. Otherwise, output the list

$$
\begin{aligned}
&\texttt{mergesortedlists}(\texttt{append}(\texttt{tail}(\texttt{foolist}, 3), \\
&\qquad\qquad \texttt{mergetwosortedlists}(\texttt{foolist}[1], \texttt{foolist}[2]))).
\end{aligned}
$$

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

$$
\begin{aligned}
&\texttt{mergesortedlists}([[-1, 2, 3], [2, 5], [1, 5, 6, 7], [-4, 0]]) \\
&= \texttt{mergesortedlists}(\texttt{append}([[1, 5, 6, 7], [-4, 0]], [-1, 2, 2, 3, 5])) \\
&= \texttt{mergesortedlists}([[1, 5, 6, 7], [-4, 0], [-1, 2, 2, 3, 5]]) \\
&= \texttt{mergesortedlists}(\texttt{append}([[-1, 2, 2, 3, 5]], [-4, 0, 1, 5, 6, 7])) \\
&= \texttt{mergesortedlists}([[-1, 2, 2, 3, 5], [-4, 0, 1, 5, 6, 7]]) \\
&= \texttt{mergesortedlists}(\texttt{append}([\ ], [-4, -1, 0, 1, 2, 2, 3, 5, 5, 6, 7])) \\
&= \texttt{mergesortedlists}([[-4, -1, 0, 1, 2, 2, 3, 5, 5, 6, 7]]) \\
&= [-4, -1, 0, 1, 2, 2, 3, 5, 5, 6, 7].
\end{aligned}
$$

Merge sort. Consider the function $\texttt{mergesort}(\texttt{foo})$, defined as follows:

- **Input:** A list `foo`.
- **Procedure:**

1. Create a list `bar` such that $\texttt{bar}[k] = [\texttt{foo}[k]]$, i.e., the kth entry of `bar` is a list of length 1 containing the kth entry of `foo`.
2. Output the list $\texttt{mergesortedlists}(\texttt{bar})$.

Problems

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

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

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

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

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