CS109A Notes for Lecture 1/24/96

Proving Recursive Programs Work

- When a program is recursive, we can often find a natural inductive proof that it works.
  - The induction is often on the number of recursive calls that must be made by a given call to the recursive function — or some equivalent parameter.

Example: Let's consider our recursive binary-converter:

```
      void convert(int i) {
(1)       if(i>0) {
(2)           convert(i/2);
(3)           putchar('0' + i%2);
          }
      }
```

Statement to be proved:

$S(i)$: `convert` produces the binary representation of integer $i \geq 0$.

- Note that ϵ is the correct binary representation of 0 in this context.

Basis: $i = 0$. The test of line (1) fails, so ϵ is printed.

Induction: Assume $S(j)$ for $0 \leq j < i$ and prove $S(i)$ for $i \geq 0$.

- Note, we are breaking our habit of proving $S(i+1)$ from smaller cases; but it doesn't matter whether we call the next case to be proven $i+1$ or i.

- If i is even, say $i = 2j$, then `convert` prints the binary representation of j at line (2) followed by 0 at line (3).
  - Appending the final 0 multiplies the value printed by 2, which gives the representation of $2j$, or i.

- If i is odd, say $i = 2j + 1$, then line (2) again prints the binary representation of j (because `i/2` throws away the remainder), and line (3) prints 1.

    - Appending 1 has the effect of multiplying the value printed by 2 and then adding 1. Again, the result is i, since $i = 2j + 1$ in this case.

Sorting

To *sort* a list = to reorder its elements so each precedes the next according to some ordering $\leq$. That is, $(a_1, a_2, \ldots, a_n)$ is sorted if

$$a_1 \leq a_2 \leq a_3 \leq \cdots \leq a_n$$

Mergesort

A recursive sorting algorithm:

Basis: A list of length 1 is already sorted.

Induction: For lists of ≥ 1 element

1. Split the list into two equal-as-possible parts.
2. Recursively sort each part.
3. *Merge* the results by repeatedly pulling the smaller element from the fronts of the two sorted lists.

- Mergesort is the preferred method for *external* sorting (sorting of lists so large that disks must be used).

Example: $(3, 1, 4, 1, 5, 9, 2, 6)$.

1. Split, say $(3, 1, 4, 1)$ and $(5, 9, 2, 6)$.
2. Sort recursively (details omitted): $(1, 1, 3, 4)$ and $(2, 5, 6, 9)$.
3. Merge: $(1, 1, 2, 3, 4, 5, 6, 9)$.

Splitting

We could split in many ways, e.g.

1. Count the list; say n elements. Put the first $n/2$ in one sublist and the remainder in the other.

2. Iteratively "deal" the elements to the two sublists, keeping track of which list gets the next element.

3. Both FCS and EMLP give a recursive algorithm that deals the elements two at a time. This approach avoids having to remember the "state" (whose turn it is).

Basis: If there are 0 elements, do nothing. If there is 1 element, give it to the first sublist.

Induction: If there are $n \geq 2$ elements, deal one to each sublist and recursively split the remaining $n - 2$ elements.

Merging

Basis: If one list is empty, the other list is the sorted result.

Induction: If neither list is empty, pick the smaller of the head elements. The result is the selected element followed by the result of merging the remaining lists.

Example:

List1	List2	Result
(1,1,3,4)	(2,5,6,9)	–
(1,3,4)	(2,5,6,9)	(1)
(3,4)	(2,5,6,9)	(1,1)
(3,4)	(5,6,9)	(1,1,2)
(4)	(5,6,9)	(1,1,2,3)
–	(5,6,9)	(1,1,2,3,4)
–	–	(1,1,2,3,4,5,6,9)