Lecture 27: Recursive Algorithms

Recursion may be used to define functions whose definition normally involves $\cdots$, to give algorithms for computing these functions, and to prove some of their properties.

Sums

Example. $1 + 2 + 3 + \cdots + n$

This is the function $f(n)$ defined by

Initial value. $f(1) = 1$
Recurrence relation. $f(k+1) = f(k) + (k+1)$

Example. $1 + a + a^2 + \cdots + a^n$

This is the function $g(n)$ defined by

Initial value. $g(0) = 1$
Recurrence relation. $g(k+1) = g(k) + a^{k+1}$

We can use this relation to prove by induction that $g(n) = \frac{a^{n+1}-1}{a-1}$ (a formula for the sum of a geometric series), provided $a \neq 1$.

Proof. *Base step.* For $n = 0$, $1 = g(0) = \frac{a^{0+1}-1}{a-1}$, as required.
Induction step. We want to prove

$$g(k) = \frac{a^{k+1}-1}{a-1} \Rightarrow g(k+1) = \frac{a^{k+2}-1}{a-1}.$$

Well,

$$\begin{aligned} g(k) &= \frac{a^{k+1}-1}{a-1} \\ \Rightarrow g(k+1) &= \frac{a^{k+1}-1}{a-1} + a^{k+1} \\ \Rightarrow g(k+1) &= \frac{a^{k+1}-1+(a-1)a^{k+1}}{a-1} \\ &= \frac{a^{k+2}+a^{k+1}-a^{k+1}-1}{a-1} \\ &= \frac{a^{k+2}-1}{a-1} \text{ as required.} \end{aligned}$$

This completes the induction.

Products

Example. $1 \times 2 \times 3 \times \cdots \times n$

This is the function $n!$ defined recursively by

Initial value. $0! = 1$
Recurrence relation. $(k+1)! = (k+1) \times k!$

Sum and product Notation

$$1 + 2 + 3 + \cdots + n \text{ is written } \sum_{k=1}^{n} k,$$

$$1 + a + a^2 + \cdots + a^n \text{ is written } \sum_{k=0}^{n} a^k.$$

$\sum$ is capital sigma, standing for "sum."

$$1 \times 2 \times 3 \times \cdots \times n \text{ is written } \prod_{k=1}^{n} k.$$

$\prod$ is capital pi, standing for "product."

Questions

27.1 Rewrite the following sums using $\sum$ notation.

- $1 + 4 + 9 + 16 + \cdots + n^2$
- $1 - 2 + 3 - 4 + \cdots - 2n$

The first sum can be written as

$$\sum_{i=1}^{n} i^2.$$

The second sum can be written as

$$\sum_{i=1}^{2n} (-1)^{i+1} i \quad \text{or} \quad \sum_{i=1}^{2n} (-1)^{i-1} i$$

or

$$\sum_{i=1}^{n} \big((2i-1) - (2i)\big).$$

Binary search algorithm

Given a list of n numbers in order

$$x_1 < x_2 < \cdots < x_n,$$

we can find whether a given number a is in the list by repeatedly "halving" the list.

The algorithm `binary search` is specified recursively by a *base step* and a *recursive step*.

Base step. If the list is empty, report 'a is not in the list.'

Recursive step. If the list is not empty, see whether its middle element is a. If so, report 'a found.'
Otherwise, if the middle element $m > a$, `binary search` the list of elements $< m$. And if the middle element $m < a$, `binary search` the list of elements $> m$.

Correctness

We prove that the algorithm works on a list of n items by strong induction on n.

Base step. The algorithm works correctly on a list of 0 numbers, by reporting that a is not in the list.

Induction step. Assuming the algorithm works correctly on any list of $< k+1$ numbers, suppose we have a list of $k+1$ numbers.

The recursive step either finds a as the middle number in the list, or else produces a list of $< k+1$ numbers to search, which by assumption it will do correctly.

This completes the induction.

This example shows how easy it is to prove correctness of recursive algorithms, which may be why they are popular despite the practical difficulties in implementing them.

A reminder of logarithms

$\log_2 n$ is the number x such that

$$n = 2^x.$$

For example, $1024 = 2^{10}$, and therefore

$$\log_2 1024 = 10.$$

Similarly $\log_2 512 = 9$, and hence $\log_2 1000$ is between 9 and 10.

Running time of binary search

Repeatedly dividing 1000 by 2 (and discarding remainders of 1) runs for 9 steps:

$$500, 250, 125, 62, 31, 15, 7, 3, 1$$

The 10 halving steps for 1024 are

$$512, 256, 128, 64, 32, 16, 8, 4, 2, 1$$

This means that the binary search algorithm would do at most 9 "halvings" in searching a list of 1000 numbers and at most 10 "halvings" for 1024 numbers.

More generally, binary search needs at most $\lfloor \log_2 n \rfloor$ "halvings" to search a list of n numbers, where $\lfloor \log_2 n \rfloor$ is the *floor* of $\log_2 n$, the least integer $\geqslant \log_2 n$.

In a telephone book with 1,000,000 names, which is just under 2^{20}, it takes at most 20 halvings (using alphabetical order) to find whether a given name is present.

20 questions

A mathematically ideal way to play 20 questions would be to divide the number of possibilities in half with each question.

E.g. if the answer is an integer, do binary search on the list of possible answers. If the answer is a word, do binary search on the list of possible answers (ordered alphabetically). If this is done, then 20 questions suffice to find the correct answer out of $2^{20} = 1,048,576$ possibilities.

Questions

27.2 Which of the proofs in this lecture uses strong induction?

ANS: The proof of correctness of the binary search used strong induction. So did the argument that we need $\lfloor \log_2 n \rfloor$ steps to search a list of length n. And the analysis of "20 questions"...

27.3 Imagine a game where the object is to identify a natural number between 1 and 2^{20} using 20 questions with YES-NO answers. The lecture explains why 20 questions are sufficient to identify any such number.
Explain why less than 20 YES-NO questions are not always sufficient.

Each question divides the remaining numbers into two categories: "Yes" and "No" for that question. Overall, k questions will divide the options into 2^k categories. If $k < 20$ then $2^k < 2^{20}$ so there must be at least one category that has at least two potential answers in it, by the Pigeon Hole Principle. Two such answers cannot be diagnosed by our questioning.