August 22, 2017

4 - The Pigeon Hole Principle and Complexity

William T. Trotter
trotter@math.gatech.edu

The Pigeon Hole Principle

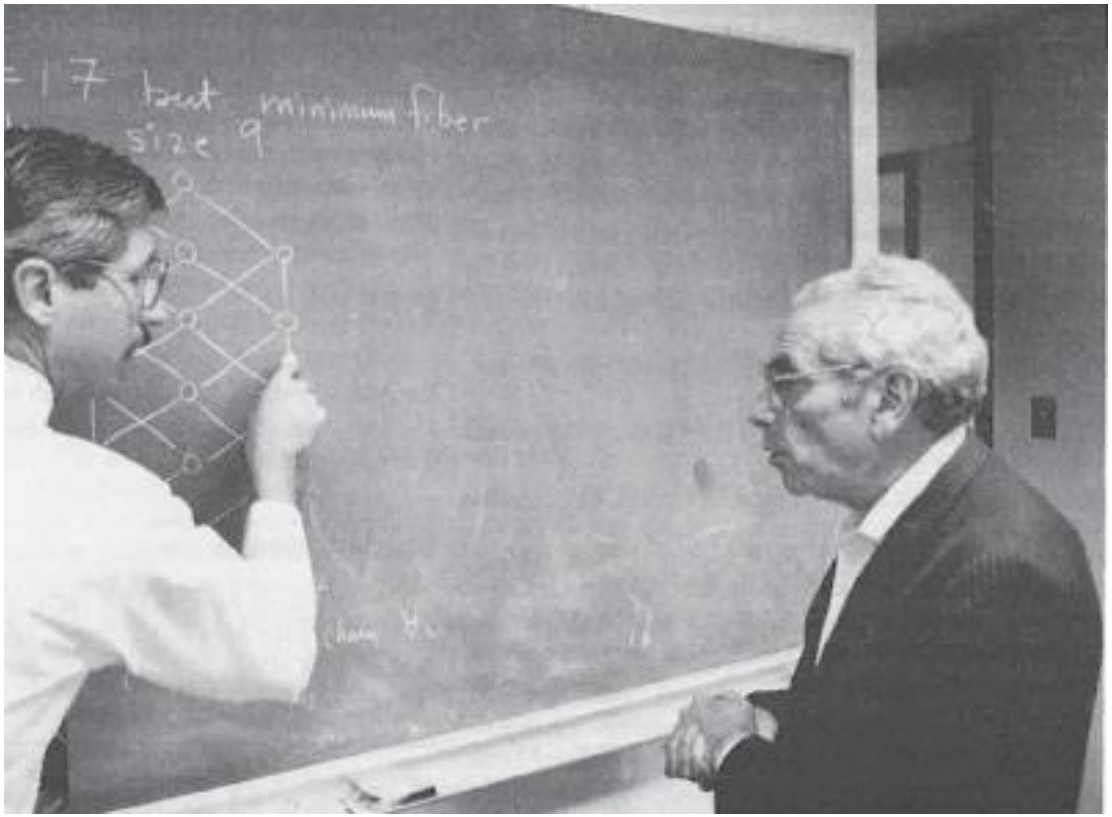
Old Saying If you have to put $n + 1$ pigeons into n holes, then you must put some two pigeons into the same hole.

More generally If you have to put $mn + 1$ pigeons into n holes, then you must put some $m + 1$ pigeons into the same hole.

The Erdős-Szekeres Theorem

Theorem Any sequence of $mn + 1$ distinct real numbers either contains an increasing subsequence of length $m + 1$ or a decreasing subsequence of length $n + 1$.

WTT and Paul Erdős (1988?)



The Erdős-Szekeres Theorem

Theorem Any sequence of $mn + 1$ distinct real numbers either contains an increasing subsequence of length $m + 1$ or a decreasing subsequence of length $n + 1$.

Example Here $m = 3$ and $n = 5$

$2, 3, -5, 0, \pi, 9, -4, -3, 7, 8, 5, 1, -6, 10, -8, -1$

Problem Find the longest increasing subsequence and the longest decreasing subsequence.

Proof of the Erdős-Szekeres Theorem

Proof Let the sequence be: $(a_1, a_2, a_3, \ldots, a_t)$ with $t = mn+1$. For each i, place the pigeon a_i in the pigeon hole (inc, dec) where inc is the length of the longest increasing subsequence starting with a_i and dec is the length of the longest decreasing subsequence starting with a_i. Then there are $mn + 1$ pigeons and only mn holes. Consider two pigeons a_i and a_j which are assigned to the same hole where $i < j$. If $a_i < a_j$, then inc value for a_i is larger than the inc value for a_j. If $a_i > a_j$, then dec value for a_i is larger than the dec value for a_j.

Easy Application

Exercise Show that if S is a subset of size 6 from $\{1, 2, 3, \ldots, 9\}$, then there are two distinct elements of S whose sum is 10.

Complexity and Problem Size

Observation We typically say that a problem size is n when the data for a problem can be interpreted as n packets of information with each packet readable in some constant amount of time. For example, when integers are at most MAX_INTEGER in some programming language, we view all integers as being readable in constant time. With this notion in mind, we begin to discuss – quite informally – the running time for algorithms to solve a problem of size n.

Running Time

Concept An algorithm accepts input data of size n and then carries out a procedure which takes a total of $f(n)$ steps. The function $f(n)$ is called the running time of the algorithm. Typically, it is not possible to determine $f(n)$ precisely, so it is very important to be able to estimate $f(n)$ in a reasonably accurate manner.

Concept It is also very important to be able to compare two algorithms, one with running time $f(n)$ and the other with running time $g(n)$.

A Small Catalog of Increasing Functions

Fact The following functions all increase and tend to infinity:

$\log^* n$ $\quad \log \log \log n$ $\quad \log \log n$ $\quad \log n$ $\quad n^{.001}$

$\sqrt{n}$ $\quad n$ $\quad n \log n$ $\quad n^{3/2}$ $\quad n^2$ $\quad n^3$ $\quad n^{\log \log n}$

$n^{\log n}$ $\quad 2^n$ $\quad (\log n)^n$ $\quad (\sqrt{n})^n$ $\quad n^n$ $\quad 2^{2^n}$ $\quad 2^{2^{2^n}}$

The Big-Oh Notation

Formally When $f(n)$ and $g(n)$ are two functions, the notation $f = O(g)$, which is sometimes written $f(n) = O(g(n))$ means there is a constant $C > 0$ so that $f(n) \leq C\, g(n)$ for all n.

The Big-Oh Notation

Example Consider two algorithms A_1 and A_2 which have running time, respectively, of $30n^2 + 150\, n \log n$ and $n^3 + n$ respectively. Which one is faster?

Clearly, when the problem size is small, the second one wins, but when the problem size becomes reasonably large, the first one is better. Formally, if we set

$f(n) = 30n^2 + 150\, n \log n$ and
$g(n) = n^3 + n$

In this case, we can write $f = O(g)$.

The Little-Oh Notation

Formally When f(n) and g(n) are two functions, the notation f = o(g), which is sometimes written f(n) = o(g(n)) means that the ratio f(n)/g(n) tends to 0 as n tends to infinity, i.e.,

$$\lim_{n\to\infty}\left(\frac{f(n)}{g(n)}\right) = 0$$

Applying the Little-Oh Notation

Example Compare the two functions:

$f(n) = 30\ n^2 + 150\ n \log n$ and
$g(n) = n^3 + n$

When n is of modest size, $g(n)$ is smaller, but from some point on $f(n)$ is smaller. In fact the ratio $f(n)/g(n)$ goes to 0. In this case, we can write

$$f = o(g).$$

Revisiting Increasing Functions

Fact In each case, if f appears before g in the following list, then $f = o(g)$.

$\log^* n$ $\quad \log \log \log n$ $\quad \log \log n$ $\quad \log n$ $\quad n^{.001}$

$\sqrt{n}$ $\quad n$ $\quad n \log n$ $\quad n^{3/2}$ $\quad n^2$ $\quad n^3$ $\quad n^{\log \log n}$

$n^{\log n}$ $\quad 2^n$ $\quad (\log n)^n$ $\quad (\sqrt{n})^n$ $\quad n^n$ $\quad 2^{2^n}$ $\quad 2^{2^{2^n}}$

Four Motivating Problems

Observation We want to develop a framework for discussing the difficulty of a problem. As examples, given a list S of n distinct positive integers, consider the following problems:

1. What is the largest integer in S?
2. If a is the first integer in S, are there distinct integers b and c in S so that $a = b + c$?
3. Are there three distinct integers a, b, c in S with $a = b + c$?
4. Can we solve the fair division problem for S?

Elementary Algorithms

Example Finding the largest integer in S can be done in n steps where $n = |S|$ is the problem size.

Example If a is the first number in S, determining whether there are numbers b and c in S so that $a = b + c$ can be done in $\binom{n-1}{2}$ steps.

Example Determining whether there are numbers a, b and c in S so that $a = b + c$ can be done in $\binom{n}{3}$ steps.

The Fair Division Problem

Example Determining whether the fair division problem can be solved for S can be done in $n2^n$ steps. Each step consists of choosing a subset T of S, and adding all the numbers in T, adding all the numbers in $S - T$ and then checking whether the answers are the same. In contrast to the other algorithms, each step takes a formidable amount of time, as $n - 1$ additions and one comparison have to be made.

Sorting

Sorting Problem You are told that there is an unknown linear order on the integers in {1, 2, …, n} and your job is to discover this order by asking questions of the form: Is $i < j$ in L? For example, if L = (2,5,3,1,4), you might discover it by asking:

Is $2 < 1$ in L? Answer Yes
Is $4 < 3$ in L? Answer No
Is $3 < 1$ in L? Answer Yes
Is $2 < 5$ in L? Answer Yes
Is $3 < 5$ in L? Answer No
Is $1 < 4$ in L? Answer Yes

The UGA Sorting Algorithm

UGA Sorting Algorithm For each 2-element subset {i, j} of {1, 2, …, n}, ask the question:

Is i < j in L?

So a total of $\binom{n}{2} = \frac{n(n-1)}{2}$ questions are asked.
Now you have all the information and can easily assemble the linear order L with these answers.

Lower Bound on Sorting

Theorem In worst case, any sorting algorithm can be forced to ask $\log_2 n!$ questions.

Explanation There are $n!$ different linear linear orders (permutations) of $\{1, 2, \ldots, n\}$. Each time a question is asked, in worst case the number of possible answers is reduced by at most $\frac{1}{2}$. So if t questions are asked, we must have $2^t \geq n!$

This is equivalent to $t \geq \log_2 n!$

The Stirling Approximation

Theorem In advanced calculus courses, the following asymptotic formula is proved:

$$\lim_{n\to\infty} \frac{n!}{\sqrt{2\pi n}\left(\frac{n}{e}\right)^n} = 1$$

It follows that $\log_2 n! \sim n \log_2 n$.

Accordingly A sorting algorithm is said to be optimal if its running time is $O(n \log n)$.

Some Optimal Sorting Algorithms

Fact The list of optimal sorting algorithms includes: merge sort, heap sort, introsort, Timsort, Cubesort and Block Sort (there are others).

An Informal Discussion of Merge Sort

Concept To determine an unknown linear order on $[n]$, first divide the problem into two equal size subproblems. That is, first find the restriction to the subset consisting of the first $n/2$ integers in $[n/2]$. Then find the restriction to the last $n/2$ integers $\{n/2 + 1, n/2 + 2, \ldots, n\}$. Then "merge" the two answers to determine the full linear order.

Fact Two sorted lists of size $n/2$ can be merged in running time n.

Fact Merge sort has a running time $r(n)$ satisfying the recurrence $r(n) = 2\, r(n/2) + n$. An easy calculation now shows that $r(n) = O(n \log n)$.