June 1981

Report. No. STAN-CS-81-863

A Programming and Problem-Solving Seminar

by

Donald E. Knuth

Allan A. Miller

Research sponsored in part by

International Business Machines

Department of Computer Science

Stanford University
Stanford, CA 94305

Computer Science Department

A PROGRAMMING AND PROBLEM-SOLVING SEMINAR

by

Donald E. Knuth and Allen A. Miller

This report contains a record of the autumn 1980 session of CS 204, a problem-solving and programming seminar taught at Stanford that is primarily intended for first-year Ph.D. students. The seminar covers a large range of topics, research paradigms, and programming paradigms in computer science, so these notes will be of interest to graduate students, professors, and professional computer scientists.

Thr productlon of this report was supported In part by the IBM Corporation.

Table of Contents

This report is a record of the autumn 1980 session of CS 204, a problem-solving and programming seminar taught at Stanford that is primarily intended for first-year Ph.D. students.

The class is organized as a discussion section in which students work on five problems. Two weeks are spent discussing **and** solving each problem (usually involving some form of computer programming in the solution).

On the following pages, each problem is presented, followed by a summary of the class discussion it provoked, followed by a summary of the solutions students came up with. Some of the discussion appears to be out of order (for example, Problem 5 appears before Problem 4), due to the fact that Problem 4 required some hardware and software resources that were not available as planned. However, the class discussions are all organized by problem number. In the interest of brevity, each participant is referred to by a two- or three-letter name. The correlation between these names and the actual students can be found at the end of the report.

Class: CS 204, "Problem Seminar". Meets 11:00–12:15 Tuesdays and Thursdays **in** room 380U (math corner, in the quad).

Discussion leader: Don Knuth. Office is MJH 328, telephone 497-4367. Meetings outside of class time are by appointment only.

Teaching assistant: Allan Miller. Office is MJH 022, telephone 497-3796. Office hours are the hour following class, plus any other time you can find me.

Textbooks: Alto and Mesa manuals, available from your friendly TA.

Purpose (from "Courses and Degrees"): Solution of various problems, numeric **and** symbolic, on a computer, using various languages. Emphasis on efficiency **of** programming, proofs of correctness, and clarity of documentation. **Presenta-**tion of solutions by students.

Actual purpose (for next year's "Courses and Degrees"): To introduce the major paradigms of computer science research,

Grading: Students should work in groups of two or three on each problem. There are five problems and we will take them in order, spending about two weeks on each. You should hand in a commented listing of your computer programs for each problem, along with a writeup describing the approaches you took to solving it. This writeup should include a discussion of what you did that worked or didn't work, and when appropriate it should also mention what you think would be promising approaches to take if there were extra time to pursue things further. Your written work will be graded on an A-E scale for your own information, but your overall grade for the course will be either **"pass"** or "nothing".

Class notes: Classroom discussions will mostly involve the homework problems, but we will try to emphasize general principles of problem solving that **are** illustrated by our work on the specific problems that come up. Everyone is encouraged to participate in these discussions, except that nobody but Knuth will normally be allowed to talk more than three times per class period. **After** class, the TA will prepare notes about what went on, so that you will be able to participate freely in the discussions instead of worrying about your own note-taking. According to department policy, these classnotes cost $1.00 **for** the entire quarter; please pay this amount to the TA as soon as you can.

Computer use: You may use the SAIL and/or SCORE and/or LOTS and/or **ALTO** computers, or anything else you can steal time on.

Caveat: This course involves more work than most other **3-unit** courses at Stanford.

Today was spent mostly in taking care of administrative details and getting to know each other. DEK explained that the course is supposed to teach research and problem-solving methods, *i.e.*, creative solutions to problems. He said that the goal of the course is to discover general techniques for problem solving rather than specific tricks. (A trick is a method that is used once; a technique is a method that can be used at least twice!) He stressed that the class will work together on problem solutions rather than in a competitive atmosphere; and in order to encourage participation by everybody, a limit of three turns at speaking per class period will be enforced.

Then DEK read some comments from the course evaluation of CS 204 the last time he taught it. The evaluations stressed that the program write-ups are one of the most important parts of the class, and that they should emphasize the methods used to solve the problems rather than the specifics of the programs. The evaluations also noted that the work load in the course is rather heavy. DEK explained that his goal is to work students at their "maximum capacity".

In closing, DEK noted that he likes to work on problems a while before reading what others have done to solve them, and when he reads about a problem he is working on he tries to guess what will be on the next page before he sees it. This is not only more fun, it makes the next page easier to understand.

Problem 1. Alphabetized Integers.

If you were to express the positive integers in English (e.g., 1 as "one", 2 as "two", and so on) and then alphabetize them, the first power of two in the list would be 8 ("eight") and the first prime number in the list would be 8,018,018,851 ("eight billion eighteen million eighteen thousand eight hundred fifty-one"). Here we are assuming that blanks and hyphens are ignored in the alphabetization, so that numbers beginning with "eight billion eight hundred thousand . . ." are larger than those beginning with "eight billion eighteen million . . .".

The problem is to find the *last* power of two and the *last* prime.

The problem of the first prime was raised by Edward R. Wolpow in Word Ways 13 (1980), 55-56, who said that it is "computationally impossible to determine the alphabetically last prime." We hope to prove him wrong.

The English names for numbers have to be defined rigorously to do this problem. We will decide on an exact rule in class. The only number names for higher powers of ten in Webster's Third Unabridged dictionary are:

10^{2} = hundred
10^{3} = thousand
10^{6} = million
10^{9} = billion
10^{12} = trillion
10^{15} = quadrillion
10^{18} = quintillion
10^{21} = sextillion
10^{24} = septillion
10^{27} = octillion
10^{30} = nonillion
10^{33} = decillion
10^{36} = undecillion
10^{39} = duodecillion
10^{42} = tredecillion
10^{45} = quattuordecillion
10^{48} = quindecillion
10^{51} = sexdecillion
10^{54} = septendecillion
10^{57} = octodecillion
10^{60} = novemdecillion
10^{63} = vigintillion

and 10^{303} = centillion. (In addition, 10^{100} is sometimes called a "googol", but we will ignore this anomaly.) That leaves us with gaps of nameless numbers $k \cdot 10^{303} + 10^{66} \leq n < (k + 1) \cdot 10^{303}$ for $0 \leq k < 1000$, and no way to name any number that is 10^{306} or more.

However, there is a natural way to continue the pattern from 10^{63} to 10^{303}: To get from one decillion to one vigintillion, note that the prefixes "un-", "duo-", "tre-", "quattuor-", "quin-", 'sex-", "septen-", "octo-", and "novem-" systematically increase the power of ten in steps of three. We can use this method along with each of the following 'suffixes":

10^{33} = decillion
10^{63} = vigintillion
10^{93} = trigintillion
10^{123} = quadragintillion
10^{153} = quinquagintillion
10^{183} = sexagintillion
10^{213} = septagintillion
10^{243} = octagintillion
10^{273} = nonagintillion

For example, the conventional name for a googol would be "ten duotrigintillion" under such a scheme. To solve this problem, we shall assume that if anyone ever invents analogous names for 10^{3n+3} with $n > 100$, these names will be come alphabetically after "billion" and alphabetically before "vigintillion".

An efficient test for primality was devised by Michael Rabin and Gary Miller; this test is not rigorous but it fails with probability $\leq \frac{1}{4}$. Therefore, if the test succeeds fifty times for a number n, the probability that n is not prime is $\leq 2^{-100}$. For practical purposes, we will say that this means n is prime, since such reliability is better than that of any computer. The test is implemented with the following algorithm:

Let $n = 1 + 2^k q$ where q is odd. Choose a random integer x in the range $1 < x < n$. Set $j \leftarrow 0$ and $y \leftarrow x^q$ modn (the remainder of x^q divided by n). Now repeat the following: If $j = 0$ and $y = 1$, or if y = $n - 1$, terminate and call n "prime". If $j > 0$ and $y = 1$, or if $j = k - 1$, terminate and call n "nonprime". Otherwise set $j \leftarrow j + 1$ and $y \leftarrow y^2$ mod n. Note that in the loop it is always true that $y = x^{2^j q}$ mod n.

Class notes for October **7.**

DEK started a discussion of the first problem by saying that even though it is a "toy problem" it can help develop the mental structures needed for computer science problems arising in real applications.

The discussion then turned to the problem of formalizing the method of naming numbers in English. We found that there is some disagreement on how certain numbers are named. For example, 3,000,144,010,000 is called "three trillion one hundred forty-four million ten thousand" in America; but RSF, from New Zealand, calls it "three trillion one hundred and forty-four million ten thousand" (he mentioned that the older British custom of calling 10^{12} "one billion" has fallen into disuse, probably even in Britain). The class decided to stick with the American system, which does not use the word "and" (which might have changed the problem considerably since

it is so alphabetically small!) and uses the suffixes listed in the problem statement. As for the task of formalizing the naming of numbers, RSF suggested expressing the numbers in "base 1000" so that a number N would be

$$N = (a,. \ . \ . \ a_2 \ a_1 \ a_0)_{1000} = \sum_{i=0}^{n} a_i 1000^i, \quad \text{where } 0 \leq a_i \leq 999.$$

With this setup, a function name(d) can be defined on $0 \leq d \leq 999$ and'then **the** name of a number N is

$$\text{American(N)} = t(n). \ . \ . \ t(0)$$

where

$$t(i) = \begin{cases} name(d_i)\, thousandname(1000^i), & \text{if } d_i \neq 0; \\ \text{null}, & \text{if } d_i = 0. \end{cases}$$

JMM suggested that finding name(d) includes three cases: $1 < d < 19$, $20 \leq d \leq 99$, and $100 \leq d \leq 999$. In the second and third cases the **definition** of $name(d)$ uses a value from the previous case. DEK formalized this by saying that

$$name(d) = \begin{cases} smallname(d), & \text{if } 0 < d \leq 19; \\ tenname\left(\left\lfloor \frac{d}{10} \right\rfloor\right) name(d \bmod 10), & \text{if } 20 \leq d \leq 100; \\ name\left(\left\lfloor \frac{d}{100} \right\rfloor\right) \text{ "hundred" } name(d \bmod 100), & \text{if } d \geq 100. \end{cases}$$

Here *smallname*, *tenname*, and *thousandname* can be combined into a single function dictionary. There was some discussion on whether or not this means that the algorithm will work for most other languages simply by changing the dictionary, but the class decided that it won't; for example, the French system is based in part on twenty rather than ten (e.g., 94 is **"quatre-vingt-quatorze"**, literally **"four**-twenty-fourteen").

The discussion then turned to methods for naming numbers larger than 10^{303}. MMS suggested that the same algorithm could be used, but powers of 10 greater than 10^{303} could be named by using more **"centillion"s**. For example, 10^{306} would be "one thousand centillion", 10^{606} would be **"one** centillion centillion", 10^{609} would be **"one** thousand centillion centillion" and so on. DEK formalized this by writing:

$$dictionary(10^{3k+3}) = \begin{cases} \text{as in class notes}, & \text{if } k \leq 100; \\ dictionary(10^{3(k-101)+3}) \text{ "centillion"}, & \text{if } k > 100. \end{cases}$$

For example, $300{,}300 \times 10^{606}$ would be called "three hundred thousand centillion centillion three hundred centillion centillion".

GMK proposed another system that split numbers into groups of 10^{303^k} using

$$American(N) = American\left(\left\lfloor \frac{N}{10^{303}} \right\rfloor\right) \text{ “centillion” } American(\text{N mod } 10^{303})$$

when $N > 10^{303}$. In this scheme $300{,}300 \times 10^{606}$ would be called “three hundred thousand-three hundred centillion centillion”. Although both schemes can be interpreted unambiguously, the class decided to use the one proposed by MMS since there is no “last number” in the one proposed by GMK; an alphabetically larger number can always be produced by adding another “centillion” onto the end of the name.

The discussion turned to finding the “last” prime and power of two given this algorithm for generating the English names of numbers. DOH mentioned that the powers of two are well-behaved, easy to calculate, and fairly sparse, whereas the prime numbers are hard to generate and fairly dense. This suggested that finding the last power of two might be done by generating powers of two “near” the last number, and finding the last prime would be done by generating numbers in decreasing alphabetical order and testing their primality. This started a discussion of how the last numbers might be generated. JP and DEK talked about a method that would generate all legal “next letters” using a finite automaton, but only use the alphabetically largest one. The class finished just as RLH suggested that it would be easier to just generate the form of the alphabetically last hundred or so numbers by hand and then use a program to test their primality.

Class notes for October 9.

DEK began by saying that someone had pointed out that the naming convention chosen last class period had the disadvantage of making it impossible to find the first prime since an arbitrary number of “centillion’? can be put in “eight billion centillion centillion . . .”. He suggested changing the dictionary function to

$$dictionary(10^{3k+3}) = \begin{cases} \text{as in class notes,} & \text{if } k \leq 100; \\ \text{“CS” } dictionary(10^{3(k-101)+3}) \text{ “centillion”} & \text{if } k > 100. \end{cases}$$

This made the official name of $300{,}300 \times 10^{606}$ “three hundred CS CS thousand centillion centillion three hundred CS centillion centillion”.

The rest of the class was spent discussing the prime-testing algorithm. The algorithm to test the primality of n is:

Find k such that $n = 1 + 2^k q$
Find a random x such that $1 < x < n$
j ← 0
$y \leftarrow x^q \bmod n$
Repeat:
  If $(j = 0 \wedge y = 1) \vee y = n - 1$
    Then terminate calling n "prime".
  Else if $(j > 0 \wedge y = 1) \vee (j = k - 1)$
    Then terminate calling n "nonprime".
  Else
    $j \leftarrow j + 1$
    $y \leftarrow y^2 \bmod n$

This algorithm gives the correct answer when it says "nonprime" and is correct at least $\frac{3}{4}$ of the time when it says "prime". It relies on two facts about arithmetic modulo a prime, namely (in this discussion p denotes a prime)

$$\text{if } x \bmod p \neq 0 \text{ then } x^{p-1} \bmod p = 1 \qquad \text{(A)}$$

which is also called "Fermat's test", and

$$\text{if } x^2 \bmod p = 1 \text{ then } x \bmod p = 1 \text{ or } x \bmod p = p - 1. \qquad \text{(B)}$$

Eq. (A) follows from the fact that the set {xmodp, $2x \bmod p, \ldots, (p-1)x \bmod p$} is just the numbers $\{1, 2, \ldots, p-1\}$ in some order; hence $x \cdot 2x \cdot \ldots \cdot ((p-1)x) = 1 \cdot 2 \cdot \cdots \cdot (p-1) = (x^{p-1} - 1)(p-1)!$ is a multiple of p, and $x^{p-1} - 1$ must be a multiple of p. To prove (B), note that $x^2 - 1$ is a multiple of p if and only if $x - 1$ or $x + 1$ is a multiple of p, since $x^2 - 1 = (x-1)(x+1)$.

The algorithm sets y to $(x^q \bmod n)$, $(x^{2q} \bmod n)$, $(x^{4q} \bmod n)$, $(x^{8q} \bmod n)$, and so on up to $(x^{(n-1)/2} \bmod n)$. Each value of y except the first is the square of the previous y. By out fact (B), any y other than the first one can only be 1 if the previous y was either 1 or $n - 1$. But the way the algorithm is set up, this cannot be true. So if we find $j > 0 \wedge y = 1$ then n cannot be prime. Furthermore, if we run through all values of y and find that $(x^{(n-1)/2} \bmod n)$ is not 1 or $n - 1$ then we know by (B) that $x^{n-1} \bmod n$ had better not be 1. But by (A), $x^{n-1} \bmod n$ must be 1, so if we reach $j = k - 1$ without detecting that $y = 1$ or $y = n - 1$, the number n cannot be prime. Although DEK didn't justify the statement with a proof, he said that if these tests for non-primality fail, the number n has a probability of more than $\frac{3}{4}$ of being prime, in the sense that more than $\frac{3}{4}$ of all numbers $1 < x < n$ will witness to the fact that a non-prime n isn't prime.

The next topic of discussion was about techniques for doing high-precision arithmetic. MWT started things off by asking if x can be generated as a single-precision (one-word) number, since all starting values of x are supposed to be equally

good. DEK replied that if the generalized Reimann hypothesis is true, then there is at least one good x in the range 1 to $4(\ln n)^2$, in which case the algorithm could be made into a deterministic one by testing n with all values of x in this range. MPH asked how many bits of precision are necessary to represent the numbers involved, and DEK estimated that about 200 bits would be needed to represent numbers near two vigintillion. If arithmetic is done using a base of 2^{17} (to prevent overflow while doing integer multiplies on the PDP-10) then 200-bit numbers will fit in an array of 10 to 15 words. The class decided that in finding $(y^2 \bmod n)$, the y^2 part is pretty easy but the modulo operation is more difficult since it requires a division. DEK mentioned that division is relatively easy to do in base two, involving only one-bit shifts and binary subtractions. FJB suggested that the shifting might be done in words rather than bits; DEK agreed, but pointed out that the program would get more complicated since a trial divisor would have to be calculated in order to avoid doing about 2^{16} multiple-precision subtractions.

The next problem was to figure out how to find $x^q \bmod n$. RSF suggested the following recursive algorithm:

```
if q = 1 then return(x modn) else
   if q is even then return(x^(q/2) mod n)^2 mod n
   else return x(x^(q-1) mod n) mod n
```

The number of "mod n" operations is then at most $2 \log_2 q$.

DOH suggested some methods for speeding up the process of finding the last prime. He said that numbers divisible by two or five can be discarded syntactically (i.e., not even generated), and divisibility by three could be easily determined by adding digits together mod three. RLH added that a sieving method can be used to remove all numbers that are divisible by small primes from the last hundred or so numbers. The big multiple-precision primality test should be used only when there is good reason to suspect that n is prime.

PEV was interested in the number of operations involved in the multiple-precision arithmetic, in order to estimate the running time of the programs. DEK did a quick analysis of the algorithm for finding $(x^q \bmod n)$. Assuming that n has ν bits and q has ρ bits and T(p) is the number of operations to find $x^q \bmod n$, then

$$T(\rho) = \begin{cases} 1, & \text{if } \rho = 1; \\ T(\rho - 1) + O(\nu^2), & \text{if } \rho > 1 \end{cases}$$

which is $O(\rho\nu^2)$. [The $O(\nu^2)$ represents one or two operations of multiplying v-bit numbers and then dividing a 2ν-bit number by a v-bit number.]

The class finished just as DEK mentioned that one of the "bad" numbers for the prime-testing algorithm (a number for which almost $\frac{1}{4}$ of the values of x return "prime" when the number is actually composite) is 1729. This number is also the

smallest number that can be represented as the sum of two cubes in two different ways.

Class notes for October 14.

Today in class about eight people said they had found the last prime and the last power of two, but they were unwilling to reveal their answers. FNY finally said his group's program had to check 23 numbers for primality and that the answer **is** the same as the alphabetically last number except the last three digits are 293. He said he had calculated the last power of two using a calculator.

The question came up of whether or not there is an algorithm to find the alphabetic predecessor of a given number. MMS said he had thought of a "simple, obviously correct method" to do so. It uses a procedure parse that determines if a string is either a valid number or a valid prefix. Each string is **a** sequence of words from a dictionary, ignoring spaces, where no word in the dictionary can. be a prefix of another word in the dictionary. Therefore, "eighty" and "eighteen" do not appear in the dictionary, but "eight", **"y",** and **"een"** do. The algorithm works as follows:

> Replace the last word in the string with its alphabetic predecessor, unless there is no predecessor. In the latter case, remove the last word; if the result is a valid number, terminate, otherwise, continue working on it. In the former case, if the result is a valid prefix, append the last word in the dictionary to the string repeatedly until obtaining something that is not a valid prefix. If the string that is not a valid prefix is a valid number, terminate, otherwise keep working on it.

A similar algorithm finds the alphabetic successor.

TCII pointed out that the problem of finding the predecessor is equivalent to looking at a tree with a branching factor of **26** where each branch corresponds to adding a letter (this is slightly simpler than a branching factor of d corresponding to a d-word prefix-free dictionary in MMS's scheme). Each node in this tree is a string that can be tested with parse and then the problem can be viewed as one of finding the previous node of a preorder traversal. CXF said that you can also think of the problem in terms of a stack-the number would be on a stack as powers of 10^3 followed by a number between 1 and 999. The dictionary of possible stack entries is sorted in alphabetical order and the last entry in the stack is replaced by its alphabetic predecessor (unless there is none, in which case the same method is applied to the next-to-last entry in the stack). Such a method, which combines the parsing with the preorder traversal, is however not "simple and obviously correct',.

DEK started discussing modulo arithmetic by explaining how parallel processors can do arithmetic with large numbers. Each processor does the same operations,

but processor i works modulo m_i. The m_i are chosen to be relatively prime so that the "Chinese remainder algorithm" can be used to combine the results in a final step to find the answer modulo M where A4 is the product of the m_i.

Next, DEK brought up the question of how badly the choice of x in the primality-testing algorithm can affect the results. Working through an example of testing $n = 1001$ (note that $1001 = 7 \times 11 \times 13$) went this way: the only values of x that cause the algorithm to incorrectly say "prime" for 1001 are those for which $x^{1000} \bmod 1001 = 1$. This is equivalent to saying that $x^{1000} \bmod ? = 1$, $x^{1000} \bmod 11 = 1$, and $x^{1000} \bmod 13 = 1$; equivalently, if $x \bmod 7 = a$, $x \bmod 11 = b$, and $x \bmod 13 = c$, then

$$a^{1000} \bmod 7 = 1,$$
$$b^{1000} \bmod 11 = 1,$$
$$c^{1000} \bmod 13 = 1.$$

Fermat's theorem says that $x^6 \bmod 7 = 1$, $x^{10} \bmod 11 = 1$, and $x^{12} \bmod 13 = 1$. It can be proved (using Euclid's algorithm) that if $x^m \bmod p = 1$ and $x^n \bmod p = 1$ then $x^{\gcd(m,n)} \bmod p = 1$. Therefore, since $\gcd(6, 1000) = 2$, $\gcd(10, 1000) = 10$, and $\gcd(12, 1000) = 4$, the only values of x that might "fool" the algorithm are those x such that $(x \bmod 7, x \bmod 11, x \bmod 13) = (a, b, c)$ where

$$a^2 \bmod 7 = 1,$$
$$b^{10} \bmod 11 = 1,$$
$$c^4 \bmod 13 = 1.$$

Since this is the "roots of unity" problem, there are 2 values of a, 10 values of b, and 4 values of c that solve these equations. Therefore there are $2 \times 10 \times 4 = 80$ values of x that might "fool" the algorithm. Since the algorithm chooses x randomly between 1 and 1001 here, it picks a bad value at most about 8% of the time. In fact, the algorithm isn't even fooled by all of these; for example if $c^2 \bmod 13 = 12$ then $x^{250} \bmod 13 = 12$ and $x^{250} \bmod 7 = x^{250} \bmod 11 = 1$, so $x^{250} \bmod 1001 \notin \{1, 1000\}$ but $x^{500} \bmod 1001 = 1$.

JJW asked why DEK had mentioned at the beginning of class that it would be desirable to know the factors of N — 1 (where N is the alphabetically last prime). DEK replied that these factors could be used to make a deterministic test for the primality of N. This test takes advantage of the fact that if

$$\{1 \bmod n, x \bmod n, x^2 \bmod n, \ldots, x^{n-2} \bmod n\} = \{1, 2, 3, \ldots, n-1\}$$

and $x^{n-1} \bmod n = 1$ for some x, then n is prime. In addition, if n is prime there are many values of x for which these statements are true. However, if the powers of x do not generate the numbers from 1 to $n-2$, then the numbers they generate will have some period d, and $n-1$ is a multiple of d. So if $n-1$ factors into $p_1^{e_1} p_2^{e_2} \ldots$ ● p;*

then it is only necessary to show that $x^{(n-1)/p_i} \bmod n \neq 1$ for $1 \leq i \leq k$ for some value of x and in addition $x^{n-1} \bmod n = 1$. This proves the primality of n. DEK mentioned that the best known way to factor N — 1 is to take out the small (single-precision) factors, then use the nondeterministic primality-testing algorithm on what is left. He said that sometimes large numbers factor into a set of small primes and one large one; we might be lucky.

The class ended with DEK presenting a factoring problem that had recently been solved. It had been known for some time that $2^{256} + 1$ is nonprime, but no one knew any of its factors. The algorithm that found one of its factors (an U-digit one) worked like this: Suppose N = $n_1 n_2$. If the series x_j is generated so that $x_{j+1} = (x_j^2 + 1) \bmod N$, and we take $y_j = x_j \bmod n_1$ and $z_j = x_j \bmod n_2$, then we will find that $y_{j+1} = (y_j^2 + 1) \bmod n_1$ and $z_{j+1} = (z_j^2 + 1) \bmod n_2$ since multiplication and addition are valid operations modulo a prime. While generating $y_1, y_2, \ldots, y_k, \ldots$, eventually a y_m is generated such that $y_m = y_k$ for some value of k and m, and the expected size of k and m is about $\sqrt{n_1}$. (This is something like hashing: if you put random numbers into a hash table of size n, two of them will almost surely hash to the same place by the time you have inserted about $\sqrt{n}$ numbers.) At this point $\gcd(x_m - x_k, N) = n_1$, unless we have the highly unlikely coincidence $z_m = z_k$, thus providing the desired factor. So the algorithm just needs to generate the sequence of x_j and check $x_{17} - x_{16}, x_{18} - x_{16}, \ldots, x_{31} - x_{16}, x_{33} - x_{32}, \ldots$ for having a GCD with N that is larger than 1. This GCD is the desired factor.

Solutions for problem 1.

Most groups solved this problem successfully, finding the "last" prime to be 2,-000,000,000,000,000,000,000,000,002,000,000,000,000,000,000,000,002,000,000,002,-293, which is pronounced '*two vigintillion two undecillion two trillion two thousand two hundred ninety three", and the "last" power of two to be 2^{320}, which is 2,135,-987,035,920,910,082,395,021,706,169,552,114,602,704,522,356,652,769,947,041,607,-822,219,725,780,640,550,022,962,086,936,576 and is pronounced "two untrigintillion one hundred thirty five trigintillion nine hundred eighty seven novemvigintillion thirty five octovigintillion nine hundred twenty septenvigintillion nine hundred ten sexvigimtillion eighty two quinvigintillion three hundred ninety five quattuorvigintillion twenty one trevigintillion seven hundred six duovigintillion one hundred sixty nine unvigintillion five hundred fifty two vigintillion one hundred fourteen novemdecillion six hundred two octodecillion seven hundred four septendecillion five hundred twenty two scxdecillion three hundred fifty six quindecillion six hundred fifty two quattuordccillion seven hundred sixty nine tredecillion nine hundred forty seven duodccillion forty one undecillion six hundred seven decillion eight hundred twenty two nonillion two hundred nineteen octillion seven hundred twenty five septillion seven hundred eighty sextillion six hundred forty quintillion five hundred fifty quadrillion twenty two trillion nine hundred sixty two billion eighty six million nine

hundred thirty six thousand five hundred seventy six".

Everyone's approach to the problem was basically the same for finding the last prime. MACLISP was used by JP/HWT/JJW, PB/RLH, FNY/ML/MMS, and JDH. Each of these groups took advantage of the BIGNUM feature of MACLISP to do infinite-precision arithmetic. The groups that didn't use MACLISP came up with some interesting ideas for doing extended-precision arithmetic. Some groups used "base 1000" numbers for the convenience of converting these numbers to alphabetic form and back. FJB/OP, JMM/GMK, and RV used "base 100,000" numbers to reduce the size of the arrays holding the numbers. CXF/PHW/PEV used "base 10" numbers, storing one digit in each array position. DOH/RSF used "base 1000" numbers for the largest power of two and 'base 2^{16}" for the largest prime.

There were some other special tricks for handling extended-precision arithmetic. FJB/OP, DOH/RSF, and JMM/GMK each found an extended-precision number mod an integer by repeatedly multiplying the current partial result by the base, adding the next digit (going from most significant to least), and taking the mod of that as the next partial result. DOH/RSF and JMM/GMK checked to make sure that x^q was greater than n before doing an extended-precision divide when finding x^q mod n. JMM/GMK suggested that a special method for squaring extended-precision numbers could be used that would only compute the inner products once (rather than twice). JMM/GMK also used floating-point division to get an approximation for their trial divisor in the extended-precision modulus routine, although they had some trouble with roundoff errors.

Generating random numbers was done in two different ways: FJB/OP, JJM/GMK, JDH, CXF/PHW/PEV, JP/HWT/JJW, and FNY/ML/MMS generated the random number x over the entire range $1 < x < n$. DOH/RSF, PB/RLH, and RV generated the random number over a limited range, about one single-precision integer.

Almost all programs removed numbers that were divisible by two or five. Many programs checked divisibility by a number of other small primes in addition. The program of FJB/OP was notable in this respect in that it checked divisibility by all primes less than 1000. JDH actually used a formula to determine the "optimal" number of small primes to check, although he checked divisibility by odd numbers rather than primes.

JP/HWT/JJW factored the number $2 \ldots 293 - 1$ and got

$$2 \cdot 2 \cdot 3 \cdot 83 \cdot 293 \cdot 4759 \cdot n_1,$$

where n_1 has no factors less than 10^7. [During winter quarter, JJW was able to find the factorization

$$n_1 = 7396423814267 \sim 1946998172413323070585001134712803889806 13,$$

and to prove that these factors were prime. (The latter was considerably more difficult than factoring n_1!) This led to a "rigorous" proof that our number 2. . . 293 is indeed the largest prime.]

The methods for generating numbers in reverse alphabetical order varied widely. CXF/PHW/PEV, JP/HWT/JJW, and RV programmed methods to generate all numbers in reverse alphabetical order; CXF/PHW/PEV used an explicit heapsort to order the names of the numbers from 1 to 999. DOH/RSF and JDH had a method that could generate the last 100 numbers in reverse alphabetical order, and FJB/OP, JJM/GMK, and FNY/ML/MMS had a method to generate the last 1000 numbers. PB/RLH had an interesting method-they tested 200 numbers close to the desired answer for primality and checked by hand to see which one was alphabetically last!

People approached the "last power of two" problem in three ways. OP/FJB, SGD/DEK2/MPH, JDH, and FNY/ML/MMS found the proper exponent of two by hand (using a hand calculator) and then used a program to find the actual number. DOH/RSF generated all powers of two that fit in their number representation and kept track of which one was last alphabetically. JP/HWT/JJW, PB/RLH, and RV checked a subset of the powers of two and kept track of which one was last alphabetically.

Problem 2. A Chess Endgame.

It is well known that a king and queen can defeat a king and rook except for **a** few unusual starting positions, However, at the IFIP congress **at** Toronto in 197'7, Ken Thompson presented a program that took the side of king and rook **in** this game. He challenged Hans Berliner, the former World Correspondence **Chess** Champion, and Lawrence Day, the Chess Champion of Canada, to demonstrate how the king and queen could win. They accepted the challenge but failed to **beat** the computer, finally giving up.

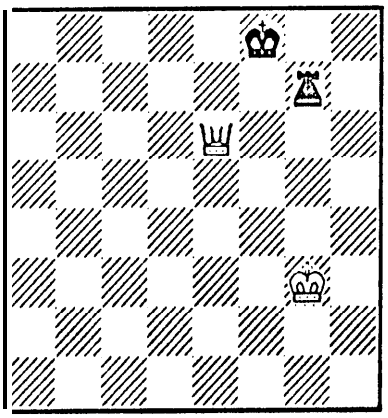
Thompson's program moved from each position **to** the next legal **one that** maximally delayed checkmate. Your problem is to do the same: write a program that plays a black king and rook against a white king and queen from any starting position, making sure that Black remains in play as long as possible.

Several things should be easy to determine as spinoffs of your program. Can you find a position that forces white to use the maximum number of moves in order to checkmate Black (assuming that each player makes the best possible move **at every** turn)? Can you count and perhaps also characterize all positions with White to move in which White *cannot* force checkmate? It will probably also be easy **to** modify your program to take the part of Berliner and Day, and do better than they did (if it is possible for White to win).

In doing this problem, we will of course ignore the "50-move" rule that **allows** the king-rook player to declare a stalemate after 50 moves have been made.

Class notes for October 16.

Discussion of problem two, the chess endgame problem, began today after the results of problem one were collected. JDH had brought in a book about chess endgames. This book stated that the king and queen vs. king and rook problem is a tricky one to play, and that the strategy for the white king and queen is to try to force Black to make a bad. move. It showed a stalemate possibility for Black, namely:



After White moves his king, Black checks again with the rook. The white king dare not move into the king file because the black rook would do the same and take the white queen on the next move. Eventually Black forces the white king to get **into** a loop or to take the black rook in the rook file; then Black is left without a move but is not in check.

FJB suggested that there are two ways to solve Problem 2: using heuristics to guide the play or enumerating every possible play. DEK said that this type of problem arises often in artificial intelligence research. It is a generalization of the "shortest-path" problem, for which positions are represented as a graph consisting entirely of "OR" nodes; in this case there are both "AND" nodes and "OR" nodes. He mentioned that Ken Thompson's solution reportedly involved the use of a database with three million entries in it, and said that this indicated that memory management would be an important part of the problem solution.

OP raised a question about the "50-move" rule in chess. There was some discussion, and the class finally agreed that the official rule has changed at least twice in recent years but in general if 50 moves are made without a pawn move or a piece being captured, either side has the option to call a draw unless the other side can prove that there is a winning strategy.

DEK pointed out that humans and computers play chess in very different ways. In this problem, the computer has access to a large data base, something a human does not have. He mentioned that John McCarthy is doing research into how humans "condense" this database internally. DOH asked if one would use a different strategy playing against a computer than against a human, and DEK replied that in general while playing chess against a computer it is a good idea to make sacrifices early if they will lead to positional advantage later on, since the computer will never miss a short-range play but current programs do not see long-range plans using general goals. DEK2 said that some psychology studies indicate that a grand master's memory of a chess position is dependent on his having played into that position; i.e., it is very difficult for humans to remember random positions.

DEK prodded the class into formalizing the method of finding the optimal solution as required by the problem statement, FNY said that the idea is to define the number of "moves to win" for any board position, and try to maximize that. OP objected that this rule didn't distinguish between whose turn it was to move, given the same board position. Finally DEK suggested defining p as a board position, W(p) as the smallest number of White moves necessary to force a win for White, and B(p) as the largest number of moves that checkmate can be delayed. The class reached a consensus that if the next legal plays by White are to positions $p_1, p_2, \ldots, p_n$, and White is to move, then

$$W(p) = \begin{cases} \infty, & \text{if } n = 0; \\ \min(B(p_1), B(p_2), \ldots, B(p_n)) + 1, & \text{if } n \neq 0; \end{cases}$$

while if the next legal plays by Black are to positions $p_1, p_2, \ldots, p_n$, and Black is to move, then

$$B(p) = \begin{cases} 0, & \text{if } n = 0 \text{ and black king in check;} \\ \infty, & \text{if } n = 0 \text{ and black king not in check;} \\ \max(W(p_1), W(p_2), \ldots, W(p_n)), & \text{if } n \neq 0. \end{cases}$$

SGD pointed out that this definition does not distinguish the case where Black checkmates White. DEK suggested that a possible solution for this might be to use something like ∞^2 for the $n = 0$ case in the definition of $W(p)$, but this point was not discussed in great detail.

DOH brought up the memory management problem at this point by asking how many different board positions exist. To set an upper bound, DEK said that a crude method is to simply represent each piece's position, and use some special encoding like putting the rook on top of the king to indicate that the rook has been taken. This met-hod requires 64^4 board positions. There was a brief discussion of how the encoding can be reduced through the use of symmetry. DEK mentioned a theorem by Burnside on symmetry: if the transformations on an object form a group with size G then the total number of inequivalent positions is

$$\frac{1}{G} \sum_g (\text{number of ways an object is unchanged under g})$$

where the gs represent the transformations. For example, in the chess endgame being considered, there are 64^4 ways the board can be unchanged under the identity transformation and 8^4 ways the board can be unchanged under reflection around a diagonal. There are eight different transformations including the identity transformation, but the other five do not have fixed points, so the total number of inequivalent boards is $\frac{1}{8}(64^4 + 8^4 + 8^4)$ which is about 2.1 million. Assuming about 6 bits per position, the program requires about 12 million bits, or about $\frac{1}{3}$ megaword on the PDP-10. The discussion ended with DEK telling about a trick for declaring large arrays in SAIL: the SAIL compiler won't allow array indices over 16 or so bits, but an array of 200,000 words can be declared as `[0:1,0:99999]` and the i^{th} element referred to as `memory[location(a[0,0])+i]`.

[Here is a proof of Burnside's theorem: Let the positions be p_1 through p_N and the transformations be g_1 through g_G. Two positions p_i and p_j are equivalent if $p_i = g_k(p_j)$ for some k. Suppose there are m classes of inequivalent positions, and assume that pi, . . . , p_m are mutually inequivalent. Then the list $g_k(p_j)$ for $1 \leq j \leq m$ and $1 < k <$ G_includes each position p_i exactly x_i times, where x_i is the number of g's such that $g(p_i) = p_i$. For example, if p_i is equivalent to p_1, say $p_i = g_l(p_1)$, then there are x_i solutions to the equation $g_k(p_1) = p_i$, since

$g_k(p_1) = p_i$ if and only if $g_k(g_l^{-1}(p_i)) = p_i$. Thus $mG = x_1 + \ldots + x_N$. But $x_1 + \ldots + x_N$ is the sum over all g of the number of positions fixed by g; hence m is this sum divided by G.]

Class notes for October 21.

Today DEK presented the idea that the chess problem can be expressed using a context-free language. Let $W(p)$ denote the set of all strings that define winning strategies for White starting at position p with White to move, and let $B(p)$ denote the set of all strings that define winning strategies for White starting at position p with Black to move. These strategies are to be given in some appropriate format; for example, we can say in pseudo-BNF that

$$(\mathrm{W(p)}) ::= \text{move p to } p_1; \langle B(p_1)\rangle \ldots \text{move } p \text{ to } p_n; \langle B(p_n)\rangle$$

and

$$\langle B(p)\rangle ::= \{p \text{ to } p_1 \rightarrow \langle W(p_1)\rangle; \ldots; p \text{ to } p_n \rightarrow \langle W(p_n)\rangle\}$$

if the legal moves from position p are to positions pi,. . . , p_n; however, we have

$$\langle B(p)\rangle ::= \text{“checkmate”}$$

if Black is checkmated in position p. If there are no legal moves from position p (i.e., stalemate), the corresponding $\langle W(p)\rangle$ or $\langle B(p)\rangle$ is not the left-hand side of any production so the corresponding language will be empty (it will contain no strings).

This grammar obviously defines the language of *all* possible winning strategies for White, starting from a given position. It is a huge grammar with millions of nonterminals, and the individual languages $W(p)$ are often infinite. So how can this help with Problem 2? Answer: the context-free formulation of this problem puts the complex circular recursions into a more familiar theoretical setting; we can use this grammar as a conceptual tool, although we would never actually construct it. JMC said that it would be helpful to think about the language in terms of a “meta-BNF” to generate the large grammar. OP suggested that the “winning moves” for White arc those that lead to positions with a winning strategy. Similarly, Black positions with a winning White strategy are those in which all possible moves are winning moves for White. RLH pointed out that there are some moves that are not in the language. For example, in some board positions White has a choice of moves where some are “winning moves” (guaranteeing a forced win) and some are not. In this case the language will not generate the non-winning moves, because the corresponding language $B(p_i)$ will be empty.

One of the interesting features of the grammar is that each language $W(p)$ it defines will be empty if and only if there is no winning strategy for White starting at position p. There is a well-known algorithm to determine whether or

not a context-free language is empty (at least DEK and HWT knew it): Start with all terminal symbols "marked" and all nonterminal symbols "unmarked". Then repeatedly mark any nonterminal symbol that produces a string consisting entirely of marked symbols, until this is no longer possible. At this point it is not difficult to prove that a nonterminal produces at least one string if and only if it is marked.

It is not difficult to find the shortest string derivable from any nonterminal symbol whose language is nonempty, using a slight modification of this emptiness-testing algorithm. If we look at what that procedure does, in the case of the chess-playing grammar, it first marks the $B(p)$ that are checkmates, then marks the $W(p)$ that are "mate in one", then marks the unmarked $B(p)$ whose only moves lead to marked $W(p)$, then marks the predecessors of these $B(p)$ (which are "mate in two"), and so on. Thus we obtain a systematic procedure that applies nicely to Problem 2. [See D. E. Knuth, "A generalization of Dijkstra's algorithm", *Information* Processing Letters 6 (1977), 1-5, for other applications of this context-free language approach to finding the shortest paths in AND/OR graphs.]

The discussion turned to specific implementation details when FJB asked if the program should find the optimal strategy for Black when Black can checkmate White. DEK said that it would probably be best to do whatever was easiest and fit into the algorithm best, rather than trying to treat the situation as a special case. RSF suggested that it might be easier to work on a smaller version of the problem, for example a 4 x 4 board. DEK said this would be a good way to test the program before using a lot of computer time doing the full 8 X 8 board, as long as no special hacks are in the program that take advantage of the board size. He said that these special hacks are often necessary to optimize the program enough to run, and usually involve either optimizing inner loops of code (since they are executed so often) or optimizing the structure of the basic elements in the data base (since they are replicated so many times). FJB suggested that an easy way to reduce the number of disk accesses (one of the slowest operations) is to split the database into sections based on the least moved piece (the white king). Since there are ten positions for this piece (using symmetry, and Burnside's theorem), the data can be split into ten "pages", one of which can be in core at any time. This started a discussion of symmetry, and OP asked whether it is known that the 8 rotation and reflection transformations are the only symmetries in this problem (for example, some other transformations might involve swapping rows or columns). DEK said that since adjacency must be preserved and corners must map into corners, only those eight transformations are symmetric. RSF asked what the definition of an "illegal move" was, and FNY said that it was any move that left the king in check. He also pointed out that this was assuming that we would be ignoring the possibility that Black's first move was to castle!

DOH suggested that one way to reduce the number of disk accesses is to keep two versions of the database, one of which is indexed by the position of the black

pieces and the other of which is indexed by the position of the white pieces. In this way, **all** possible moves for Black can be examined without any disk accesses, and after a move is made the white database can be accessed to examine all possible white moves.

The class finished just as RSF mentioned that it is necessary to generate all possible checkmates and stalemates in order to start a "bottom-up" algorithm. PB said that it is only necessary to generate the checkmates since the database can be initialized with each entry having an infinite number of moves to checkmate, and when the program finished the unchanged entries are moves leading to stalemate.

Class notes for October 23,

DEK started the class by saying that the context-free language approach he used last class was only a formalism for thinking about the problem and shouldn't be thought of as an actual implementation of a solution. He also mentioned that by calling disk accesses "one of the slowest operations" he meant random-access operations in which the disk head had to move, since an actual disk transfer is usually less than ten times as slow as a memory transfer once it gets started.

JMC then gave a small talk on his ideas about the problem. He feels that the problem can be done in one 256K core image. The important thing to do is to avoid representing illegal positions and store only "significant" positions. He had used a "Monte Carlo" method to estimate the number of significant positions and had found that about $\frac{1}{4}$ of all possible positions are significant. He suggested using the position of three of the pieces to index a table of pointers that are either zero (for the "insignificant" positions) or point to a table in which every entry had 6 bits indicating the fourth piece position and 6 bits to hold the count of "moves to checkmate" (this final table would need to be searched sequentially). He also mentioned that the division of 6 bits (one piece) is rather arbitrary; the final table could have entries of 5 bits of the fourth piece and 6 bits for the count, or 7 bits (storing partial information about the third piece) of piece information and 6 bits of count, and so on. He suggested that effective use can be made of a space/time tradeoff in two ways. Only even counts need to be stored, since the "operating program" that uses the tables to play against an opponent can use a 4-ply lookahead to find the best move in the table.

He then made a very interesting suggestion that the program can keep two bit-maps indexed by every possible position. One is the "White-to-move" table and one the "Black-to-move" table. At the start of the program the positions in the "Black-to-move" table corresponding to illegal positions or losses for Black would be marked (i.e., set to one). Then all possible backward moves for White would be generated and these positions would be marked in the "White-to-move" table (the "White-to-move" table marks positions in which White has a sure win). Following that, the "Black-to-move" table would be marked by starting from every position

and marking positions where every possible move leads to a position marked in the "White-to-move" table. By repeating this cycle, all the "mate in *n*" positions for White can be found, and the table of counts can be updated.

He suggested a speedup for the program: in the inner loop, while checking moves, it would be faster to check to see if queen moves would change the count (i.e., they made fewer moves to mate) before checking their legality, since legality-checking would be more complicated and the count-changing test would eliminate most of the moves right away. He estimated that the look-ahead needed to generate the "Black-to-move" table would be a lo-instruction loop and would therefore execute about 500,000 positions × 25 average moves per position × 10 instructions or 125 million instructions, about 2 minutes of CPU time on the PDP-10.

DEK and JMC had a small discussion summarizing the main points. DEK said that the hack for illegal positions probably doesn't work quite right for stalemate, and MMS said that there is a problem of having kings take kings while moving into check. DEK asked if there is any good solution for checking the legality of queen moves, and FNY suggested making a small chessboard in memory and simulating stepping the queen through the move while checking for collisions with other pieces.

MPH asked about how symmetries could be handled. DEK suggested using two octal digits ij to represent a piece position, and generating the symmetrical positions with ij, $\bar{i}j$, $i\bar{j}$, ji, $\bar{j}i$, $j\bar{i}$, and $\bar{i}\bar{j}$ where $\bar{i}$ is the binary one's complement of i. JMC pointed out that the program can be done in two passes; one pass generates the "to-move" tables and outputs them to disk or tape sequentially, and another pass reads them back in to generate the "count" table (since this is a sequential operation). DEK said that it might even pay to keep another bit table of "just-marked" positions. He thought that the graph of the number of bits marked in the table as a function of how many moves to mate would be very interesting (the graph of "ways to mate in *n*" as a function of n). JDH said he thought the graph would have a peak at about 8 moves since after capturing the rook White could usually force a mate in about 8 moves. The topic of doing tricky bit manipulations came up, and DEK asked how to find the rightmost bit turned on in a word. AAM suggested ANDing the word with its two's complement to get only that bit turned on, and DEK said that the actual bit position can be determined from a table lookup indexed by the result of taking the output of the AND modulo 37 (since the first 36 powers of two have different remainders mod 37).

DEK asked if there are any problems associated with White's "backward moves" in JMC's scheme. FNY pointed out that the backward moves have to be able to "uncapturc" a piece. JDH mentioned that it is illegal to make a backward White move that puts Black into check, since this means that it was White to move with Black in check (an illegal position).

F JR suggested that instead of doing a forward search from the "Black-to-move" map, a set of counts can be kept for each position; each count telling how many

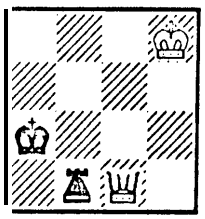
ways Black has to escape from check when playing from that position. Every time a winning move for White is found, the count of all positions that are backward moves from that winning position are decremented (the counts are initially set **to** the number of ways Black can move in that position). DEK suggested **a way to** calculate how much running time this approach saves: Assuming that each **forward** move is checked in a random order and the forward move-checking stops as soon **as** a winning white position is found, then if Black has 20 moves from one position, the number of forward moves checked when only one winning white position has been found is $21 \times \frac{1}{20}$. After two winning white positions have been found the average number of forward moves checked drops to $21 \times \frac{1}{19}$. The total average **number of** forward ***moves*** checked when building the entire database is

$$21\left(\frac{1}{20}+\frac{1}{19}+\ldots+\text{a>}\right) \approx 21(\ln 20+\gamma-1) \approx 54.6$$

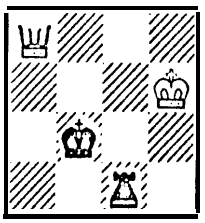
where γ is Euler's constant. This result is to be compared to the 20 **counter-**decrementing operations done using **FJB's** method.

Class notes for October 28.

Today FNY claimed to have a working program for a 4 x 4 chess board. His program output indicated that the maximum n for which White has a **"mate** in n**"** is $n = 20$. The graph of "mates in n" as a function of n starts at 213, drops down, peaks at 330 with $n = 4$, then falls off to 4 at $n = 20$. One of the **"mate** in 20" positions is



and one of the "draw" positions is



FNY also ran his program on a 6 x 6 board. It found a maximum value of 23 for n, with the "mate in n" curve having local maxima at $n = 1$, $n = 6$, and $n = 17$. FJB suggested that the peak at 6 probably corresponded to forced mate after a rook capture, as JDH had conjectured in the previous class meeting. RSF conjectured that if captures were eliminated the second maximum of the curve (at $n = 17$) would be eliminated.

FNY was not the only person with partial results. DOH said his program had finished calculating "mate in 1" positions on an 8 X 8 board and had found about 10,000, taking about 5 minutes of CPU time. JMC said that in talking to Donald Michie he had learned that the maximum value of n on an 8 x 8 board was 31. The class was excited to hear this (since it meant that counts fit in 5 bits), but became less excited when JMC revealed that he wasn't sure if Michie meant moves to mate or moves to rook capture. He also mentioned that Michie had worked on the king and rook vs. king and knight endgame.

JMC asked if anyone had made any estimates on the execution time of their program. FNY reported that his 4 x 4 program had taken 20 minutes and his 6 X 6 had taken 2 hours, but these figures were real time rather than CPU time. JMC said he thought the whole 8 x 8 case should only take about 10 minutes of CPU time to run. He was interested in the execution time because the program seems to have a non-constant "inner loop": at the beginning of execution the program spends most of its time generating backward moves for White since not many positions are marked and the forward moves for Black usually find an unmarked position soon, but toward the end of execution most of the time is spent generating forward moves for Black and skipping over marked positions.

DEK explained how Monte Carlo methods can be used to estimate the size of a tree. If, while randomly picking a path down a tree from the root to a leaf, the branching factors of the nodes encountered are n_1 (at the root), n_2 (at the first node encountered), and so on up to n_k (at the parent of the leaf), then the expected number of nodes in the tree is

$$N = 1 + n_1 + n_1 n_2 + n_1 n_2 n_3 + \cdots + n_1 n_2 \ldots n_k.$$

By doing this process several times, a good estimate of the order of magnitude of the size of the tree can be obtained. PB asked if the commonly quoted figure of 10^{120} for the size of the chess-game tree was obtained in this way. JMC said that the figure was probably just guessed at by using the average number of moves available at any position in tournament play and the average game length and assuming the tree is well-balanced and has a fairly constant branching factor.

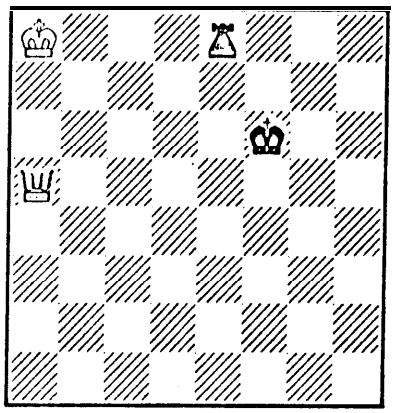
Class notes **for October 30.**

No one had any operational programs to play the chess endgame today, but HWT and GMK said they were almost done with the 8 X 8 board and that the

graph of "mate in n" appeared to have two local maxima, one at $n = 1$ and the other at $n = 7$. They mentioned that it was hard to tell if their program was correct, and DEK said that it is always difficult to debug game programs, especially heuristic ones, since the bugs are usually "covered up" later on by the program's heuristics, before an actual move is made. He said that the famous chess program by Greenblatt was probably the first debugged one. The class discussion turned to Problem 3 (which appears later in this report).

Class notes for November 4.

Today was an exciting class. We connected the department's video projector to a terminal so that the entire class could watch the proceedings, then we called Ken Thompson from Bell Labs in Murray Hill and pitted some of our programs against his. The first game we played was with the program of JDH and JJW playing White against Ken's program playing Black on the board



with White to move. (Ken's program said that this starting position resulted in the longest possible game.) The game proceeded as follows:

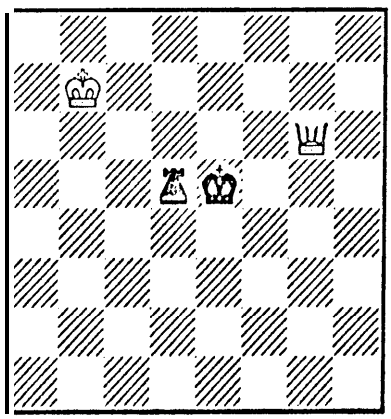
1. K-N7	R-K2 ch
2. K-B8	R-K1 ch
3. K-Q7	R-K2 ch
4. K-Q8	R-K5
5. Q-Q5	R-K4
6. Q-Q3	K-K3

At this point it was obvious to everyone that these moves were anything but obvious! Ken remarked that this was true in general; the "best" moves never seemed to follow any intelligent pattern, but both programs agreed that these were the best moves. The next three moves, according to Ken, were "forced" (in the sense that there was

only one best move for White):

7. K-B7	R-QB4 ch
8. K-N7	R-Q4
9. Q-N6 ch	K-K4

At this point the board looked like



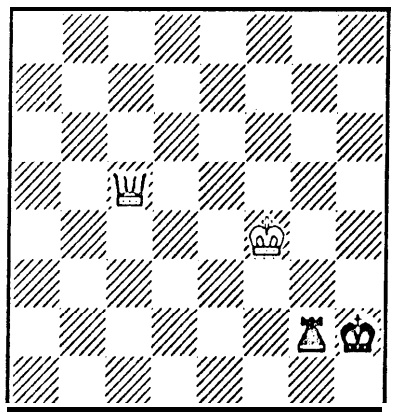
The next few moves actually had some structure, since White was forcing Black to the side of the board. Ken remarked that this play was the first time he had seen any "systematic" moves so far away from the checkmate.

10. K-B6	R-Q5
11. Q-N5 ch	K-K5
12. K-B5	R-Q6
13. Q-N4 ch	K-K6
14. K-B4	R-Q7
15. Q-N3 ch	K-K7

The play continued with some more non-obvious moves:

16. K-B3	K-B8
17. Q-R3 ch	R-N7
18. K-Q3	K-B7
19. Q-B5 ch	K-N8
20. K-K3	R-N6 ch
21. K-B4	R-N7
22. Q-B5 ch	K-R7

Now the board was

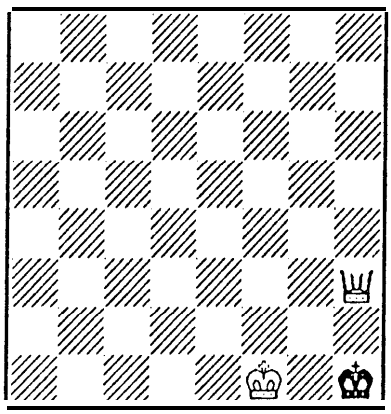


When the program of JDH and JJW made the next move, Ken said they had lost one move (they could have had a win in one less move). Not only that but the reply of Ken's program caused JDH and JJW to say that Ken could have delayed a loss by one more move! A little discussion brought out the fact that the program of JDH and JJW was trying to minimize the number of moves to checkmate, but Ken's program was trying to maximize the delay to loss of the rook. The result was that each program had a slightly different "optimal" strategy.

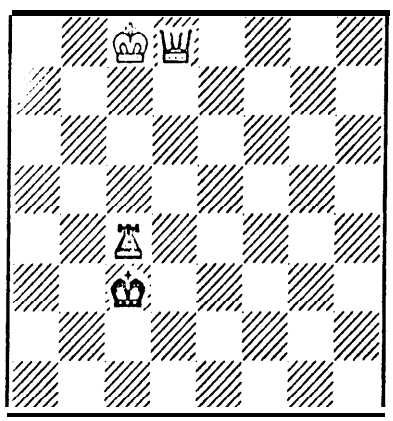
23. Q-Q5 ?	R-N6 ?
24. Q-K5	R-N7
25. K-B3 ch	K-R8
26. Q-R5 ch	K-N8

JEB remarked that the position was now a 'book" position he had seen before, and that checkmate would happen fairly soon. The program of JDH and JJW reported that Black could only last at most 7 more moves, but the next move Ken's program made (in order to save the rook as long as possible) was a "blunder" that reduced the count to 6. The result was a strange checkmate in which the rook was taken very late.

27. Q-R4	R-N3 ?
28. Q-Q4 ch	K-R8
29. K-B2	R-N7 ch
30. K-B1	R-N6
31. Q-R8 ch	R-R6
32. Q×R mate	



There were still about fifteen minutes left in the class, so GMK and HWT decided to give their program a shot against Ken's. Ken asked if anyone had found any "interesting" drawn games (unlike the standard one) but no one had seen one. The class decided to start from a standard **drawn** position



with GMK and HWT taking Black and Ken taking White. The game went pretty much as expected, although in several places the programs disagreed on the best move again. We didn't finish the game out after about 20 moves, since it became clear that Black could force a draw.

Solutions for problem 2.

The most successful approach to the chess endgame problem was the one used by FJB/OP/JP, JDH/JJW, and RSF/DOH in which two tables are kept: a "mate in *n* or less" table and a "next black escape" table. Both tables are indexed by board position (symmetrical board positions have the same index). The "mate in *n* or less" table is a one-bit table; setting the bit for a particular position during pass *n* of the algorithm means that white can force a checkmate in *n* moves or

less. The "next black escape" table entry for a particular position has a number indicating one of the 22 possible black moves that will take black from that position to a position that can postpone checkmate for at least *n* more moves. One "step" of the algorithm backs up one black move from each position marked in the "mate in *n* or less" table. If the "next black escape" in the resulting position still leads to a position not marked in the "mate in *n* or less" table, the algorithm continues (since the "escape" still works). Otherwise, the algorithm looks for the next **legal** black move (in the sequence of 22) that leads to a position not marked in the "mate in *n* or less" table. If one is found, its move number is inserted in the "next black escape" table, otherwise all positions that can lead to the position by a white move are marked as "mate in *n* or less".

By saving the positions that are marked in the "mate in *n* or less" **table on** each pass of the algorithm, enough information exists to have either white or black play optimally with only a one-move lookahead. However, JDH/JJW also saved B(p), the number of moves black can delay checkmate. This allowed them to play black with no lookahead.

All three groups had small but interesting twists in the algorithm. FJB/OP/JP handled symmetrical positions by flipping the board to one of the "standard" positions before applying the algorithm. This simplified their data structures considerably (they had ten disk files that corresponded to tables for the ten different positions of the white king), but it also resulted in large amounts of "special-case" code that was very repetitive. JDH/JJW used LEAP data structures (part of the **SAIL** language) in a very clever way to generate moves with very little code, and they also made good use of macros (another **SAIL** feature) to deal with repetitive code. JDH/JJW also had an interesting way to test their program: they used **a** random number generator to print out a sample of the database and hand-check it. RSF/DOH incorporated "heuristics" to speed up the test for checkmate, e.g., they made sure the white queen was present and the black king was either on the edge or in a corner before making more elaborate tests for checkmate. They tested their program by playing it against JDH/JJW in the iongest possible game.

FNY/MMS/ML used an algorithm with three one-bit tables indexed by board position. One of these, the "marked" table, tells which positions lead to white checkmate in less than *n* moves with white to move. Another, the "queue" table, tells which positions lead to checkmate in exactly *n* moves, again with white to move. A third, the "black to move" table, tells the positions where one or more black moves lead to positions marked in the 'queue" table. One pass of the algorithms marks all positions in the "marked" table that are marked in the "queue" table. Then it sets up the "black to move" table by making backward black moves from the marked positions in the 'queue" table. It then makes one pass through the "black to move" table to find the positions where all forward black moves lead to positions marked in the "marked" table (positions where black **has no escape).**

When it finds one of these positions it goes back one white move and marks the resulting positions in the "queue" table if they aren't marked in the "marked" table.

FNY/MMS/ML had a great deal of trouble with getting PASCAL to deal with the large arrays they were using, and decided to work with a 6×6 board. Their program took advantage of the similarity between forward and backward moves (the only difference being the distinction between capturing and "uncapturing"). They also used one "heuristic" (as did RSF/DOH), but didn't explicitly mention it-namely, checking that the black king was on the edge of the board before making more elaborate tests for checkmate.

PEV/PHW/CXF kept two values for each board position, the minimum number of moves for white to checkmate (which would be used by white) and the maximum (which would be used by black). They used the scheme suggested by JMC for indexing board positions in which only legal positions are kept in the table and indexing the table is done with a binary search. Rather than sequencing the set of searches for moves by procedural calls, they kept an explicit queue of "things to do", where doing the task at the beginning of the queue could add more to the end of the queue.

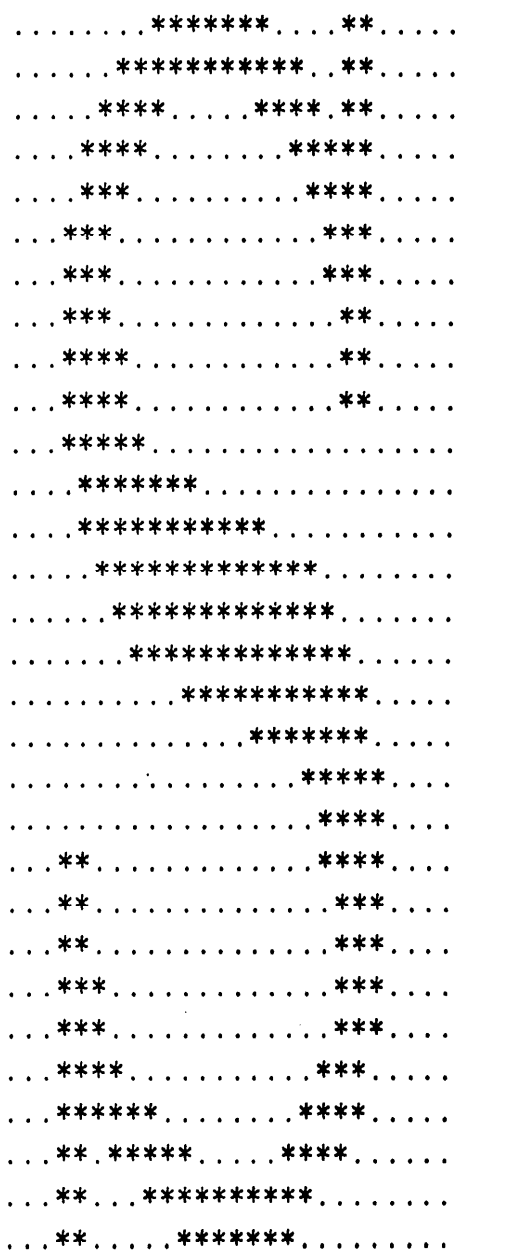
RLH/PB decided to spend some time trying to encode piece positions in a clever way to make move computations reduce to simple arithmetic. They didn't have much success, but they did make good use of SAIL macros to greatly simplify the coding of the repetitive tasks in computing moves and testing for checkmate.

GMK/HWT used an algorithm practically identical to that of FNY/MMS/ML, but they finessed the problem of dealing with large arrays by coding in C and using a VAX computer, which has a very large virtual address space. Their "play" program (which they used in the demonstration) was really just a simple program for displaying parts of the database. In order to get the next move they would display W(p) from their database and pick the move (manually) with the smallest value.

Several of the groups had a lot of trouble getting finished and many mentioned the difficulty of testing the programs for obscure bugs. In general, people felt that this was an interesting, challenging, "real-world" problem.

Problem 3. Unrounding.

This is a problem that arises in connection with digitized alphabetic characters such as the ones used to print the page you are reading. We are given the boundary of such a character as a sequence of "king moves" forming a closed circuit. **For** example, the boundary of the letter **"S"**:



is described by the following king moves (starting in the upper right corner and going counterclockwise): W, S, S, SW, NW, NW, W, NW, W, W, W, W, W, W, **SW,** W, SW, SW, S, SW, S, S, S, S, S, SE, S, SE, SE, SE, E, E, SE, E, E, E, SE, E, E, SE, SE, S, SE, S, S, S, SW, SW, SW, SW, W, W, W, W, NW, W, NW, W, NW,

NW, N, NW, N, N, W, S, S, S, S, S, S, S, S, S, E, N, N, NE, SE, E, SE, E, SE, E, E, E, E, E, NE, NE, E, NE, N, NE, N, N, N, N, N, N, NW, N, NW, NW, NW, W, W, NW, W, W, W, NW, W, W, NW, NW, N, NW, N, N, NE, NE, NE, NE, E, E, E, E, SE, E, SE, SE, SE, S, SE, S, S, E, N, N, N, N, N, N, N, N, N.

We arc actually interested in characters with a much higher resolution than this particular character, for example, characters that are 300 pixels high. We would like to find a smooth curve or set of curves that will "round" the boundary so that we can extend discrete characters to infinite resolution. In particular, if a boundary curve comes from a straight line, we would like to deduce what one such straight line is,

The boundary can first be broken into consecutive blocks such that in each block all of the king moves have only two adjacent directions; for example N and NE, or N and NW, or W and NW, etc. By symmetry let us assume that we are dealing with a section of the boundary such that all of the moves are E and NE. These points can be graphed as

$$(1, a_1), (2, a_2), \ldots, (n, a_n)$$

where $a_{k+1} = a_k + (0 \text{ or } 1)$ for $1 \leq k \leq n$.

All of this is background for the following interesting computational problem: Given a sequence of integers $a_1, a_2, \ldots$, determine the largest n such that there exists a cubic polynomial f(z) with

$$a_k - 0.5 \leq f(k) \leq a_k + 0.5, \qquad 1 \leq k \leq n.$$

After solving this problem, we will apply it to data from actual digitized characters.

Class notes for October 30.

Discussion on problem three began with DEK saying that the solution to it would probably be qualitatively different from the solution to problem two in that Some clever, easily programmed algorithm would do the trick and the interesting part is to find the most elegant algorithm. He suggested attacking the problem by first considering the linear case instead of cubics. Consider straight lines of the form $y = mx + b$ where $0 \leq m \leq 1$. In this case, the rounded points are

$$y_k = \lfloor mk + b + \tfrac{1}{2} \rfloor.$$

The "king move" description consists entirely of N and NE moves. The problem statement is simply, "Find all m and b that could have generated the given data." For example, $m = \frac{1}{3}$ and $b = \frac{1}{7}$, having $k = 0, 1, 2, 3, 4, 5, 6, 7, 8$ gives $y_k =$

0, 0, 1, 1, 1, 2, 2, 2, 3. In general, if m is rational the pattern repeats with a cycle equal to fhc denominator of m.

Next, DEK tried using $m = \psi = 1/\phi \approx .61803...$ and $b = 0$. The number ψ is interesting in this problem in that we want the fractional part of $k\psi$ for integral values of k. It turns out that $\psi^2 + \psi = 1$, so we know that $\psi^3 + \psi^2 = \psi$. Subtracting the first equation from the second gives $2\psi - 1 = \psi^3$, so the fractional part of 2ψ is ψ^3. Similarly, using the fact that $\psi^4 + \psi^3 = \psi^2$ we can find that $2 - 3\psi = \psi^4$, so the fractional part of 3ψ is $1 - \psi^4$. Continuing in this way, we find that plotting the fractional part of $k\psi$ always causes a new point to fall in the largest remaining interval, and it cuts this interval in the golden ratio. The range of $k = 0, 1, 2, 3, 4, 5, 6, 7, 8$ gives $y_k = 0, 1, 1, 2, 3, 4, 4, 5$.

The constraints on m and b can be written

$$-\tfrac{1}{2} \le b \le \tfrac{1}{2} \qquad [k = 0]$$
$$\tfrac{1}{2} \le m + b \le 1\tfrac{1}{2} \qquad [k = 1]$$

$$4\tfrac{1}{2} \le 8m + b \le 5\tfrac{1}{2} \qquad [k = 8]$$

JJW said that since this is a set of linear equations, linear programming methods can be used to solve it. JMM suggested that one might be able to combine the inequalities to further constrain m and b. OP said that the problem might lend itself to a binary search for m and b. DEK decided to take these suggestions one at a time, starting with JMM's. By combining the first two inequalities, one gets a bound of $0 \le m \le 2$. Unfortunately, this was already known. FNY suggested that using inequalities for more widely spaced values of k results in tighter bounds on m. DEK replied that although that is true, it would be nice to have an algorithm that treats values of k sequentially and knows when the constraints on m and b are no longer satisfiable if the next value of k is considered. He mentioned that Motzkin had developed a variable-elimination method procedure in the 1930s that can be used to solve linear inequalities. For example, we can write

$$\tfrac{1}{2} - b \le m \le 1\tfrac{1}{2} - b$$
$$\tfrac{1}{4} - \tfrac{1}{2}b \le m \le \tfrac{3}{4} - \tfrac{1}{2}b$$
$$\tfrac{1}{2} - \tfrac{1}{3}b \le m \le \tfrac{5}{6} - \tfrac{1}{3}b$$

and so on. For some value of m to exist, all of the left hand sides must be less than or equal to all of the right hand sides. Conversely, if all the left hand sides are less than all the right hand sides, a value of m will exist. Therefore we can remove m and obtain about n^2 inequalities involving b alone. In general,

$$-\frac{1}{k}b + \frac{y_k - \frac{1}{2}}{k} \le m \le \frac{y_k + \frac{1}{2}}{k} - \frac{1}{k}b$$

so for all values of j and k the inequality

$$-\frac{1}{j}b + \frac{y_j - \frac{1}{2}}{j} \leq \frac{y_k + \frac{1}{2}}{k} - \frac{1}{k}b$$

must be satisfied. This is equivalent to

$$\left(\frac{1}{k} - \frac{1}{j}\right)b \leq \frac{y_k + \frac{1}{2}}{k} - \frac{y_j - \frac{1}{2}}{j}$$

and since the equation should be symmetrical with respect to j and k,

$$\left(\frac{1}{j} - \frac{1}{k}\right)b \leq \frac{y_j + \frac{1}{2}}{j} - \frac{y_k - \frac{1}{2}}{k}.$$

Combining these two equations gives

$$\frac{y_k - \frac{1}{2}}{k} - \frac{y_j + \frac{1}{2}}{j} \leq \left(\frac{1}{k} - \frac{1}{j}\right)b \leq \frac{y_k + \frac{1}{2}}{k} - \frac{y_j - \frac{1}{2}}{j}$$

for all j > k. Therefore a necessary and sufficient condition for the existence of m and b is that

$$\max_{1 \leq k < j \leq n} \frac{y_k - \frac{1}{2}}{k} - \frac{y_j + \frac{1}{2}}{j} \leq \min_{1 \leq k < j \leq n} \frac{y_k + \frac{1}{2}}{k} - \frac{y_j - \frac{1}{2}}{j}.$$

Now the class started discussing the binary search approach, using a graphical method to think about the constraining inequalities. Considering the b-m plane with b horizontal and m vertical, the inequality $-\frac{1}{2} \leq b \leq \frac{1}{2}$ limits the region of solution to a vertical band of thickness 1 centered about the m axis. Each successive inequality intersects this region with another band sloping to the left; each successive band has a shallower slope than the previous ones. JDH suggested that all vertices making up the polygonal region of intersection can be found by the following method: if a vertex is contained within the current intersecting band, it remains as is, otherwise it is replaced by two vertices which are along the lines emanating from it. However, JMM provided an example of where this method wouldn't work, namely the case where the band intersects only the top of the polygonal region and loses a lot of vertices. SGD objected to the use of a binary search algorithm, since it is not obvious whether m is too large or too small if the current band doesn't intersect the polygonal region. Although this problem was not completely resolved in the class discussion, the class agreed that it would be possible to decide if a particular value $m = m_0$ was too large or too small by looking

at the interval where $m = m_0$ intersects the polygon; m_0 is OK if the successive intersections by bands $1, 2, \ldots$ does not make the interval disappear, while m_0 is too large if the kth band hits the line $m = m_0$ strictly to the left of the interval determined by bands $1, 2, \ldots, k-1$.

RLI-I made a meta-comment that the discussion of the linear case was not really addressing the problem as stated, since the stated problem was to find a cubic rather than linear equation, and that some of the methods being discussed were only applicable to two-dimensional spaces. DEK said that it is true that the two-dimensional case won't always show you what to do for the four-dimensional case, but it *will* show you what not to do, and it is easier to draw two-dimensional pictures on a blackboard than it is to draw four-dimensional ones.

DEK finished by presenting a small addendum to the properties of $y_k = \lfloor mk + b \rfloor$ when $m = \psi$. Using the recursive relation $S_n = S_{n-1}S_{n-2}$ where $S_0 =$ a and S_1 = b (for example, S_2 = ba, S_3 = bab, S_4 = babba, and so on), the kth letter of any string is $\lfloor (k+1)\psi \rfloor - \lfloor k\psi \rfloor$, with a suitable assignment of "a" and "b" to 1 and 0. Another unusual property of these strings is that in any string there are only $k+1$ different substrings of length k. Yet another property is that S_{n-1} & $_{a-2}$ is the same as $S_{n-2}S_{n-1}$ with the last two letters "complemented" (i.e., "a" changed to "b" and "b" changed to "a"). Probably the most interesting property of these strings in terms of problem three is that if "a" means "take an E step" and "b" means "take a NE step", then all of these strings correspond to drawing a line of slope ψ on a raster.

Class notes for November 6.

Discussion continued on the unrounding problem today. DEK said that some test data is available for the characters "C", "S", "2", "0", and "4" in the file CS204.DAT[MF,DEK] at SAIL or <CSD.MILLER>CS204.DAT at SCORE. In this data, the 8 king moves arc represented by single-digit integers, where 0 means "east", 1 means "northeast", and so on up to 7 being "southeast".

Throughout the discussion today, one idea that constantly arose was the question of how the unrounding program would be used. There are actually at least two ways it can be used: (1) a "data reduction" problem in which the raster form of a character is converted into a set of simple curves describing the character (this is like the METAFONT program in reverse), thereby having a compact encoding scheme; (2) a '(t-shirt" problem in which the raster form of a character is used to generate the same character on a much finer raster or with essentially infinite precision, as if we wanted to make large characters to print on a t-shirt.

SGD suggested that perhaps the problem could be solved using the fairly simple approach of representing the desired cubic function parametrically (as z(t) and y(t)) and connecting together groups of four points using splines. DEK said that this method doesn't seem to help in the data reduction problem, and in the

t-shirt problem it might produce some strange results. For example, cubic splines go through all knots, so a cubic spline fit to the points resulting from drawing a straight line into a raster would be a wiggly line through all the raster points. Even B-splines, which are affected by the knots but don't necessarily pass through them, would probably fit a wiggly line to the raster points of a straight line.

Along the lines of the data reduction problem, DOH observed that only one bit is needed to encode each step of a gradual curve in one octant. SGD suggested that an appropriate data representation (not using cubics) is a fixed-size word with a count of the number of steps to follow, followed by the octant number (3 bits), followed by a stream of bits telling the steps for drawing the curve. DEK said he believed cubics might be able to encode the data in fewer bits, especially where there are long pieces of boundary that can be encoded as one cubic; but this remains to be proved, and the value of cubics for data reduction would stand or fall based on the results the class finds for Problem 3. SGD said the division into octants seemed rather artificial and would hurt even the cubic encoding scheme when a smooth curve went, from one octant to another, since the switch would cause a new cubic to be started. He suggested that the problem could be solved with a parametric representation of the curve. DEK said he had once thought so too, and that this in fact was one one of the problems in CS204 two years ago, but no really good solution had been found.

DEK mentioned that it is fairly easy to produce hardware that generates cubics on raster points. If we have

$$f(k) = a + bk + c\binom{k}{2} + d\binom{k}{3}$$

then

$$\begin{aligned}\Delta f(k) &= f(k+1) - f(k)\\ &= b + c\,k + d\binom{k}{2}\end{aligned}$$

because

$$\begin{aligned}\Delta\binom{k}{n} &= \Delta\frac{k(k-1)\ldots(k-n+1)}{n!}\\ &= \frac{(k+1)k\ldots(k-n+2)}{n!} - \frac{k(k-1)\ldots(k-n+1)}{n!}\\ &= \frac{k(k-1)\ldots(k-n+2)((k+1)-(k-n+1))}{n!}\\ &= \frac{k(k-1)\ldots(k-(n-1)+1)}{(n-1)!} = \binom{k}{n-1}.\end{aligned}$$

Similarly,

$$\Delta^2 f(x) = c + dk$$
$$\Delta^3 f(x) = d$$

so if we have three hardware registers A, B, and C with

$$C = f(k)$$
$$B = \Delta f(k)$$
$$A = \Delta^2 j(k)$$

then we can move to the next step (from $j(k)$ to $j(k+1)$) by setting

$$C \leftarrow f(k+1) = f(k) + \Delta f(k) = C + B$$
$$B \leftarrow A\ j(k+1) = A\ j(k) + \Delta^2 j(k) = B + A$$
$$A \leftarrow \Delta^2 j(k+1) = \Delta^2 j(k) + \Delta^3 j(k) = A + d.$$

In particular, note that all three additions can be done in parallel. SGD pointed out that this method also works with parametric equations using six registers rather than three. DEK said that one of the problems with using a parametric representation is that sometimes the locus of solutions for the equation $ax^3 + bx^2y + \cdots = 0$ crosses itself. The problem is that the function $ax^3 + bx^2y + \cdots$ is positive on one side of the locus of solutions and negative on the other side, and at the crossing point an unusual arrangement of positive and negative occurs, causing algorithms to lose track of what direction they are moving along the locus. This does not happen with quadratics, however, so parametric quadratics might turn out to be better than cubics.

MMS said that an interesting problem to consider is how to fill in a character given its outline. DEK said that one method that works is to go through every point visited on the raster and complement it and all the points to the right of it. MMS pointed out that this method doesn't work on horizontal lines, and DEK confessed that it has to be hacked a little bit to get everything to work out right. One correct rule is to complement to the right of the bit you are moving to, if the move is N, NE, or NW; to complement to the right of the bit you are moving from, if the move is S, SE, or SW; and to do nothing if the move is W or E.

FJB asked what a good method would be to handle the problem of corners, *i.e.*, how to fit together the cubics making up different sections of the letter boundary. PHW said that corners are no problem in the data reduction algorithm, but would have to be considered for the t-shirt algorithm. DEK suggested that it might be useful to know how many points can be represented by a cubic starting at each point in the character boundary. Someone suggested that a good method might be to generate cubics that include some overlapping points, then to use a very small cubic

to generate the path through the overlapping points. HWT said that this method would be bad for corners, since sometimes including too many points in a cubic would change it from a very straight one to one with a slight but noticeable curve. DEK agreed and said that corners would either have to be deduced automatically or specified in the input, for the t-shirt application.

Class notes for November 11.

Today PHW asked a question about linear programming and DEK spent most of the class talking about how linear programming works.

To simplify the discussion, we shall consider the linear case of the unrounding problem, since the generalization to cubic and higher cases will be straightforward once the linear case is understood. In the linear case we have

$$f_1 \le a + b \le f_1 + \delta$$
$$f_2 \le 2a + b \le f_2 + \delta$$
$$f_3 \le 3a + b \le f_3 + \delta$$

and so on. Let us introduce "slack variables" s_i and t_i, so that these inequalities can be rewritten as three pairs of equalities

$$f_1 + s_1 = a + b = f_1 + \delta - t_1$$
$$f_2 + s_2 = 2a + b = f_2 + \delta - t_2$$
$$f_3 + s_3 = 3a + b = f_3 + \delta - t_3$$

where $s_i \ge 0$ and $t_i \ge 0$. Actually, since $s_i + t_i = \delta$, one of them could be eliminated, but it is convenient to use both in discussing the algorithms for linear programming. If we subtract the first equation from the second, we get

$$a = f_2 - f_1 + s_2 - s_1;$$

and if we subtract the second equation from twice the first, we get

$$b = 2f_1 - f_2 + 2s_1 - s_2.$$

If we let s_1 and s_2 be independent variables, then we can use the fact that $t_1 = \delta - s_1$ and $t_2 = \delta - s_2$ and make the following table:

	$-a$	$-b$	const	$-s_1$	$-t_1$	$-s_2$	$-t_2$
a	1	0	$f_2 - f_1$	1	0	−1	0
b	0	1	$2f_1 - f_2$	−2	0	1	0
s_1	*	*	*	*	*	*	*
t_1	0	0	δ	1	1	0	0
s_2	*	*	*	*	*	*	*
t_2	0	0	δ	0	0	1	1

In this table, each row gives the multiplying factors for the column headers; the sum of those products is zero. Thus, each row of the table corresponds to a linear equation. The rows for s_1 and s_2 are marked with stars since they are the independent variables.

The idea of linear programming is to maintain such a tableau subject to two invariant conditions. First, only two columns, corresponding to the independent variables in this two-dimensional application of linear programming, will have nonzero entries, except that all the other columns will contain '1' in the row corresponding to their column header. Second, all rows will be lexicographically positive; in other words, the leftmost nonzero entry will be positive. In such cases, there exists a solution to the system of inequalities, since there will be a solution to all of the equalities if we assign zero to the independent variables.

The algorithm proceeds by adding multiples of one row to another row, or appending new rows corresponding to new inequalities that need to be satisfied, while trying to maintain the two invariants. Columns a and b will contain zero in all but the first two rows, since variables a and b will never be selected as independent variables.

Let's try now to append two more rows to the tableau, corresponding to the equations $f_3 + s_3 = 3a + b = f_3 + \delta - t_3$. After subtracting appropriate multiples of the a and b rows, so that columns a and b of the two new rows are zero, the tableau looks like this:

	- a	$-b$	const	$-s_1$	$-t_1$	$-s_2$	$-t_2$	$-s_3$	$-t_3$
a	1	0	$f_2 - f_1$	1	0	-1	0	0	0
b	0	1	$2f_1 - f_2$	-2	0	1	0	0	0
s_1	*	*	*	*	*	*	*	*	*
t_1	0	0	δ	1	1	0	0	0	0
s_2	*	*	*	*	*	*	*	*	*
t_2	0	0	δ	0	0	1	1	0	0
s_3	0	0	$2f_2 - f_3 - f_1$	1	0	- 2	0	1	0
t_3	0	0	$\delta - 2f_2 + f_3 + f_1$	-1	0	2	0	0	1

Now, as long as both of the new rows are lexicographically positive, the two previously mentioned constraints are satisfied. If, however, one of them is negative (it cannot happen that both arc negative, since their constant terms sum to δ), then that row must be "fixed up" so that the invariant is re-established.

The row that needs to be fixed up can be of two types: (a) Its constant term is negative, and it has nonnegative entries in all other columns. Then there is no solution to the system of inequalities, since the algorithm has deduced a linear combination that cannot possibly sum to zero. (b) It has a negative entry in some

independent-variable column. In this case, it is possible to replace that variable with another one, as follows: Let's say that one of the independent variables is v_0, two of the dependent variables are v_1 and v_2, and we wish to make v_1 an independent variable and v_0 a dependent variable. Let v_3 be the other independent variable. Before the transformation, the table looks like

	const	$-v_0$	$-v_1$	$-v_2$	$-v_3$
v_0	*	*	*	*	*
v_1	c_1	a_1	1	0	b_1
v_2	c_2	a_2	0	1	b_2

except that the columns might be in a different order. The v_1 row means that $c_1 - a_1 v_0 - v_1 - b_1 v_3 = 0$. Dividing by a_1 gives

$$\frac{c_1}{a_1} - v_0 - \frac{1}{a_1} v_1 - \frac{b_1}{a_1} v_3 = 0,$$

which we can put into the v_0 row. Similarly, we can subtract a_2 times this equation from the equation in the v_2 row, obtaining

$$c_2 - \frac{a_2}{a_1} c_1 + \frac{a_2}{a_1} v_1 - v_2 - \left(b_2 - \frac{a_2}{a_1} b_1 \right) v_3 = 0.$$

So the new table is

	const	$-v_0$	$-v_1$	$-v_2$	$-v_3$
v_0	$\frac{c_1}{a_1}$	1	$\frac{1}{a_1}$	0	$\frac{b_1}{a_1}$
v_1	*	*	*	*	*
v_2	$c_2 - \frac{a_2}{a_1} c_1$	0	$-\frac{a_2}{a_1}$	1	$b_2 - \frac{a_2}{a_1} b_1$

In order to satisfy the constraint that the new row v_0 be lexicographically positive, it is necessary and sufficient that a_1 be positive. In order for the new row v_2 to be lexicographically positive, the condition is more interesting. Let $\succ$ denote the relation 'lexicographically greater than', let 0 denote a row vector of all zeroes, and let r_1 and r_2 denote old rows v_1 and v_2. Then $r_2 - (a_2/a_1) r_1 \succ 0$ if and only if either $a_2 \leq 0$ or

$$a_2 > 0 \text{ A } r_2/a_2 - r_1/a_1 \succ 0.$$

The latter relation is equivalent to $r_2/a_2 \succ r_1/a_1$. Note that it is impossible to have $r_2/a_2 = r_1/a_1$ because of the entries in columns $-v_1$ and $-v_2$.

In view of this theory, the algorithm fixes up a lexicographically negative row v_2 as follows: Find a column v_0 corresponding to an independent variable such that $a_2 < 0$. Look through all rows v_1 having an entry $a_1 > 0$ in this column, and choose the row having lexicographically minimum value when divided by a_1.

Make v_1 an independent variable instead of v_0. This transformation preserves the invariants, except perhaps in the row v_2 that was bad to start with; but since $a_2 < 0$, this bad row has gotten lexicographically greater than it was, so things are improving. Indeed, a finite number of such steps will either fix the bad row or produce an unfixable row. (The algorithm cannot loop, because if it chooses the same independent variables it produces the same tableau, yet the bad row is getting lexicographically larger at each step so it cannot return to a former value.)

DEK mentioned that the data structure can of course be squeezed down from a square array quite a bit since all the zero columns can be deleted and since row t_i can be deduced from row s_i without storing them both. The big array was used in the derivation only to make the proof simple.

He mentioned that most programs for linear programming use a simpler invariant: instead of the lexicographic business, they only require the constant term to be nonncgativc. Such algorithms can get into a loop, but the conditions for looping arc so rare that they are ignored. He feels that this is dangerous practice, and noted that the lexicographic test can be implemented in such a way that it runs essentially as fast as the simpler test of minimum c_1/a_1 in most cases. The problem data $f_k = k + (k \bmod 2)$ and $\delta = 1$ will cause this algorithm to produce many rows with constant term zero, and looping might well be possible. HWT suggested that a better idea might be to keep a count of how many times the process was being repeated, and to ask for manual intervention if the count became too large. DEK said that you could get away with that when solving Problem 3, but not if you were producing software for use by other people than yourself.

Both FNY and CXF brought up the question of roundoff error. CXF mentioned that this kind of approach (subtracting a multiple of one row from another) is a bad way to solve a set of linear equations, and that an LU decomposition is a much better way. DEK said that the roundoff problem is even worse than usual because a small roundoff error can completely change the results of a comparison against zero, thus changing the whole course of the algorithm. FNY suggested that since the process starts with all integers and the only nonintegral operation is divison, the arithmetic could be done with rational arithmetic (saving numerator and denominator). Someone objected that keeping the fractions in lowest form would be difficult, but DEK said that it depends on how fast the denominators grow; perhaps fractional arithmetic wouldn't be too bad. He mentioned that he was always surprised that no one had implemented Euclid's algorithm (for finding the GCD of two numbers) in hardware.

TCH mentioned that the problem of solving large linear programming problems has been studied extensively. He offered two references: R. E. Gomory, "Large and non-convex problems in linear programming" in Proceedings *of* Symposia in Applied *Mathematics* 15 (AMS, 1963) and R. E. Gomory, "Mathematical Programming",

Amer. Math. Monthly 72 (2) Part V, pp. 99-110, Feb. 1965.

Class notes for November 13.

Discussion finished on the unrounding problem today as DEK mentioned a typical solution to the roundoff problem: the "step-by-step" method can be used for a while, followed by a matrix inversion using more stable methods to solve for the current independent variables. The whole tableau is determined by the choice of independent variables, so accumulated rounding error will disappear. The only trouble occurs if your independent variables lead to a tableau that violates the invariant, in which case you have to go back or fix things up.

The class then turned to the "real-world" problem of deciding whether or not the ALTOs (which appeared to have no hardware support) should be used for Problem 4, or if communicating processes would be simulated either in a single program or using multiple jobs on SAIL or SCORE. The class opted for using the ALTOs, and AAM braced himself for the task of getting a MESA program running.

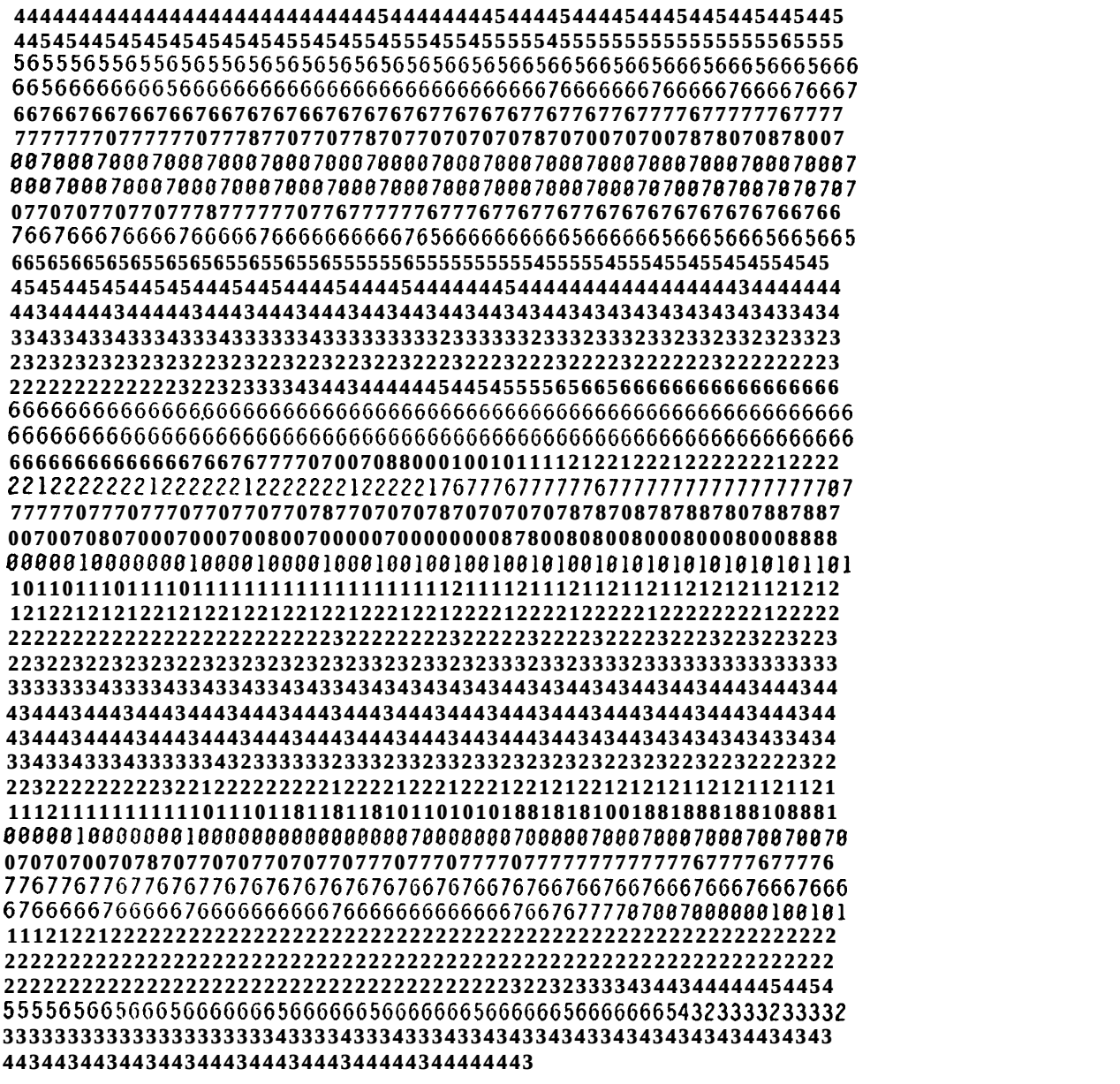
In the meantime, class discussion started on problem five, the outerplanar graph problem. (This discussion appears later.)

Test Data for Problem 3

The letter C...

```
4444444444444444444444444444444454444444444544444445444445444454
4445444454445444544544454454454454454454454454454454454454454454
5454545454554545454554554554554554554554554555545555545555545555
5555555555555555565555555655555655556556556556556556556556556565
5656565656556565656565656565665656656566566566566566566566566566
6656666566566656666566666566666566666665666666666666566666666666
6666666666666666666666667666666666667666666766666766666766666766
6766676667666766766766766766766766766766766766766766766766766766
7767676767677676767767767767767767767767767767767767777767777777677777
7777777777777770777777077770777707770770770778778778778778787078
7707070707070700707070780707007007078870078870078708878078807
0007000070000700807080007000008070800000007888000000880880888888
000000000000~0000800010000818081000180018018010010818010188101 01
0101010101011010110101181118110111181111110111111111111111111211
1111211112111211121121121121212112121212121212121212121212121212
1212212212122122122122122122122122122122122122122122222222223223233
3343443444444544545555656656656656656656656656656656656656566565
6656656565665656565656565656565655656556565565565565565556555655556
5555556555555555545555554555545545555455455455455455454545454545445
4544544544544544544544544544454444444454444444444444444344444444
3444443444434444344434434434434434434343434343434343343343433434
3343334334333433333433333334333333333233333332333332333332332333
2332332323323232332323232323232323232232232232232232232232232232
2322322232232232232232232232232232232232232232232232232232222223
2222222222222322222222222222222222222222222222212222222222222122222
2221222222212222212222212222122221222122221222122122212212212212
2122122121221221212212212121212121212121121212112121121121121111
2112111121111211111112111111111011111118111181110118111011018110
1011010101010100181010010810010010808188801800018000818888888
0010000000000000000070000000070000078087808708078870070070870 7
0707070770707707707770777077077707777787777777776777777677776
7776777677677677676776767676767676767676767676766767676676676667
6676676676667666766766666766676676767777078070008001081011112122
1222222222222222222222222222222222222222222222222222222222222222
2222222222222222222222222222222222222222222222222222222222222222
2222222222222222222222222222222232323333434434444445445455556566
5665666665666665666665666656666656666656666656666656666565332333
2333233332333333233333333333333333343333334333334334333343343343
4334343434343434343443443443443443443443443443443443444434444444
43
```

The letter S...



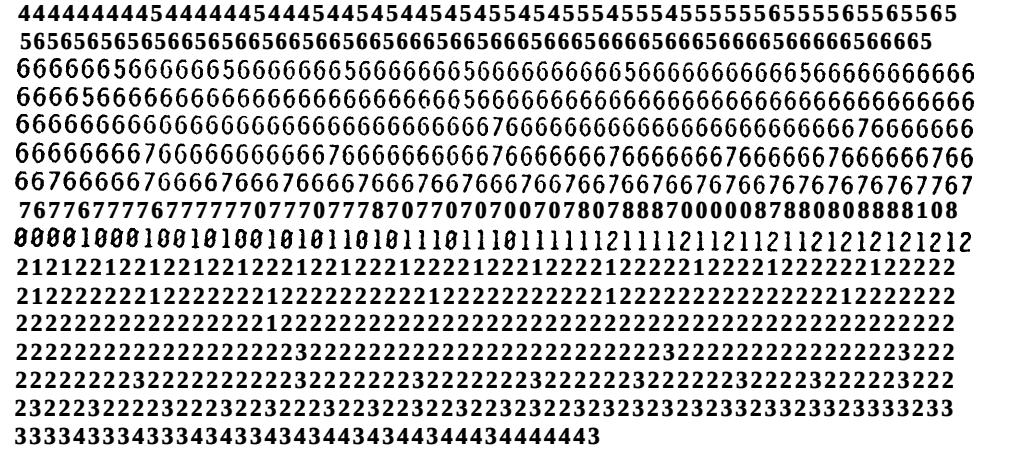
The numeral 2...

```
4444444444444444444444444454444444544445444454454445445445445445
4544545454545455454554554555545555555555555555555555655556555655
6565565656565656565656656566566566566566566566566566566566666566
6666666666666667666676676676767767777777777077070700700700007000
0000000010000100100101011011111111112112121221221222212222222222
2322223223223232332333333333433434434434434444344444444321212121
2121121211211121111211111111111011110110118118181101010108101001
0100010010000100001080000010000800000007880887000788700787870778
7077077770777777777767777677677676767767676767767667676766766676
6766766766676667666766667666766666667666666666676666666666666666
6666666566666666665666666566665666656666566566566566566566566565
6656565665656565656565556565655656555655565555655555555555555555
5565555555555555455555554555545555455455545554555455455455455445
4554554554554555455545555545555554555555555555555555555655555565
5556555655655656556565656565656565656565656566565656566566566565
6665666566656666666676676777707007000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000
0000000000000000000010010111121221222222222212222212222212222212
22221222221~22221222212222212222212222212222212222212222212222122
2221222221222212222212222222232232333343443444444544545555656656
6566666566666566666566666566665666665666665666665666665666665666654
4444444444444444444444444444444444444444444444444444444444444444
4444444444444444444444444444444444444444444444444444444444444444
4444444444444444444444444444444444444444444444444444444443111
2111112111111121111111110111~11110111110111011101110110111011011011
0110101101011011010110110110110110110110111011011101110111011110111
1011110111111101111111111011111111111111111121111111111121111121111
1211121112112111211212112121121212121212121212121212212212212212212
2122122212221222212222221222222222222222222222222222223222222223222
2223222232222232232223223223223223223223232323232323232323232333233
2333233323333233333333333333333333343333343334334334334334343434343
4343434344343443443443443444344434443444344444344444443
```

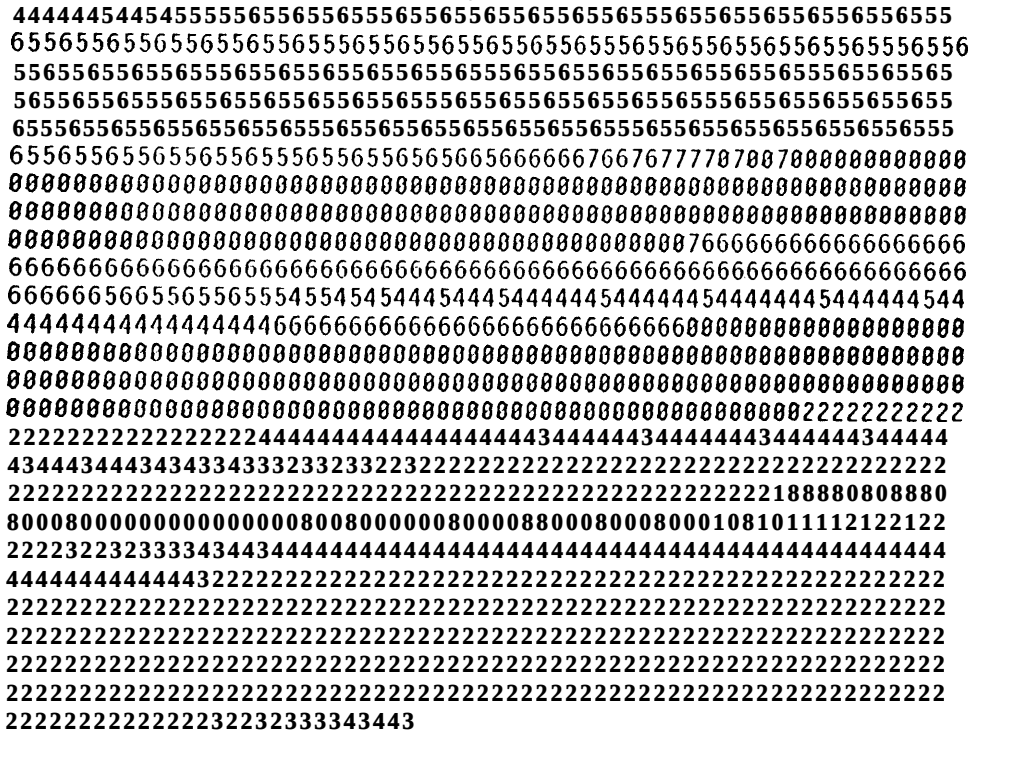
The numeral 0... outer boundary:

```
4444444444444444444444444445444444454444544454454445445445445445
4454545454545454545545545545545555555555555555555555655556556556
5565565656556565656566565656566566566566566566566566566566566566
6566656665666656066566666566665666666566666656666665666666566666
6666566666666666665666666666666666666566666666666666666666666666
6666666666666666666666666666666666676666666666666666666666666666
7666666666766666666766666676666667666667666676666676666766766766
6667666766766676676676676676676676676767676767676767676767767676
7767767767777677777777777777777777707770777077077077070770707070
7070070700700700700070007000007000007000000070000000000000000000
0000010000000010000100001001000100100100101001010101010101010101
0110111011101111111111111111111111211112112112112121121212212122
1212121221212212212212212212212212212212212212212212212212212222
2212222122222122222212222221222222221222222222212222222222212222
2222222222222221222222222222222222222222222222222222222222222222
2222222223222222222222222222232222222222223222222222232222222322
2222322222232222232222322223222322322322322322322322322322322232
2322322322322323223232322323232323232323323323323333333333333333
3333333333333433343334334343343434343434343434434434434434434434
4344443444434444444 3
```

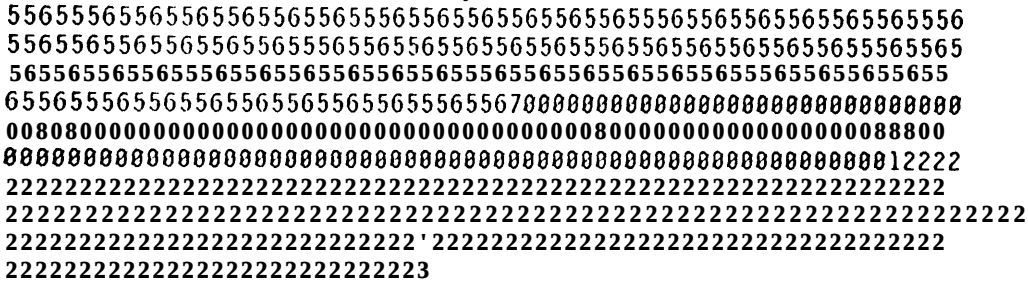
The numeral 0... inner boundary:



The numeral 4... outer boundary:



The numeral 4... inner boundary:



Solutions for problem 3.

Almost everyone used the method of linear programming discussed so extensively in class. However, CXF found the Tschebysheff approximation for a cubic polynomial to solve the equations

$$\begin{aligned}
a_3x_0^3 + a_2x_0^2 + a_1x_0 + a_0x_0 - D &= y_0 \\
a_3x_1^3 + a_2x_1^2 + a_1x_1 + a_0x_1 + D &= y_1 \\
a_3x_2^3 + a_2x_2^2 + a_1x_2 + a_0x_2 - D &= y_2 \\
a_3x_3^3 + a_2x_3^2 + a_1x_3 + a_0x_3 + D &= y_3 \\
a_3x_4^3 + a_2x_4^2 + a_1x_4 + a_0x_4 - D &= y_4
\end{aligned}$$

Adding more equations to this set increases the value of D. When $D > .5$ the last equation added is removed and the resulting set of points has been fit with a cubic. CXF found that this method would fit cubics to long linear sections of data but it usually only fit between four and six points on curved sections.

GMK/HWT used a method conceptually similar to the one discussed in class, but since they wrote their program before the class discussion, they ended up solving a "dual" problem phrased in terms of a minimization rather than a maximization. Adding a constraint to the original problem corresponds to adding a variable to the dual problem.

PB/RLH used the method discussed in class, but rather than having an algebraic derivation they used a geometric derivation where each constraint formed a "slice" bounded by two hyperplanes in the space of coefficients for the cubic.

One problem that all groups had to deal with was the reality of using finite-precision rational arithmetic. Most groups used a comparison with a small ϵ rather than directly comparing with zero. JMM/RV/SGD used a value of δ that was slightly less than one-this probably has the same effect but has a slightly different implementation. JMM/RV/SGD and PHW/PEW/CXF both had "infinite-precision' rational arithmetic versions in which they kept an integer numerator and denominator for each number, but both groups had problems with the numerators and denominators becoming very large.

Another problem that everyone had to address was the problem of cycling. Most groups ignored the problem successfully (apparently the test data didn't exercise any cycling problems). FJB/OP/JP speculated that even if their program would have cycled theoretically (they got some "zero pivots"), they might have been saved by roundoff error. GMK/HWT had a slightly different problem, since the "ratio test" was not needed to solve the dual problem. However, they needed some arbitrary ordering of variables to prevent a cycling problem, so they used the order in which the variables were added to the system. RSF/DOH dealt with cycling by putting a limit on the number of pivots that could be used before the program **would give**

up trying to add a point. However, this method seemed to be detrimental to the algorithm, because increasing this limit from ten to twenty increased the number of points fit by one cubic quite a bit.

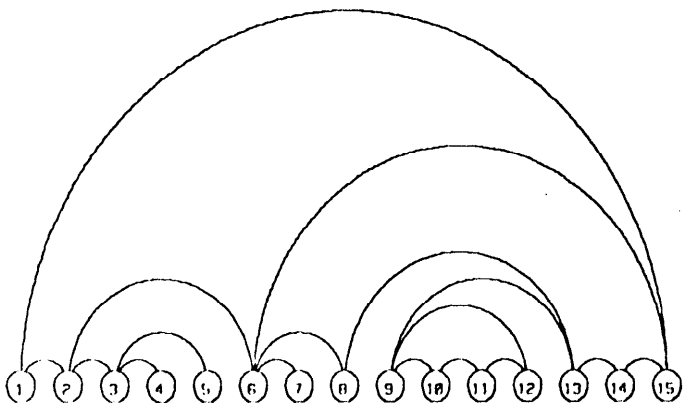
ML/MMS/JJW improved the data compression by interpolating groups. of three different king moves (for example, northwest, north, and northeast) instead of only two. This meant that they were fitting cubics to functions which could go down as well as up. GMK/HWT had a slightly different scheme: for example, if one cubic started in a string of moves moving east and ended in a string of moves moving northeast, the next cubic was not forced to end when the northeast moves ended. Instead, they allowed the cubic starting in the northeast moves to continue into north moves.

JMM/RV/SGD mentioned that the data 'compression" achieved by their algorithm was very poor—it actually expanded the amount of data by a factor of about three. However, ML/MMS/JJW paid more attention to "squeezing the bits" and came up with a compression scheme which reduced the number of characters in a file describing the data by a factor of about three over the original king-move data. In addition, they made an effort to find how many bits were significant in the mantissa of the cubic coefficients, by truncating the mantissas until they didn't satisfy the constraints on the cubic equations. Although it seems as though truncating any nonzero parts of the mantissa would result in constraints not being satisfied, they found that in many cases the mantissas could be truncated quite a bit!

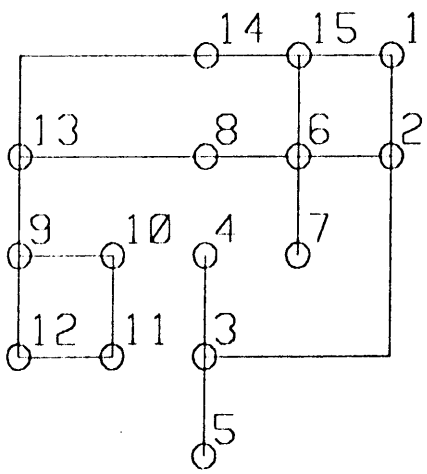
Problem 5. Grid Layout.

An "outerplanar graph" can be defined as a set of vertices $1, 2, \ldots, n$ and **a** set of distinct edges $(a_1, b_1), (a_2, b_2), \ldots,$ (a,, b_m), where $a_i < b_i$ for all i and we never have "crossing" edges such that $a_i < a_j < b_i < b_j$. A free tree can always be represented as an outerplanar graph, but there are many outerplanar graphs that are not trees; thus outerplanar graphs are more general than trees but less **general** than planar graphs.

The problem we wish to solve is to embed a given outerplanar graph in **a** rectangular grid in the following sense: The vertices $1, 2, \ldots, n$ will be placed at points of the grid, and there will be paths along edges of the grid from point a_i to point b_i for $1 \leq i \leq m$; at most one path can go through any one edge of the **grid.** For example, one of the ways to lay out the graph



is to use



We will assume that no vertex is the endpoint of more than four edges in the given outerplanar graph, since only four edges can enter one gridpoint.

In practical applications (VLSI design, for example), it is desirable to find such embeddings that fit in a rectangle containing the smallest area. The bounding rectangle in the previous example is 5×5, giving an area of 25 squares; this is not the best possible embedding.

Try to find a method that yields a small area in a reasonable amount of time. It is probably intractable to find the absolutely smallest area for a given graph, but you should use a method that attempts to come close.

Note: It is rumored that somebody has proved it possible to embed outerplanar graphs of n vertices in rectangles of area $\leq cn$ for some constant c. Can you discover such a proof too? It may be helpful to consider maximal outerplanar graphs, i.e., outerplanar graphs in which no further edges (a_{m+1}, b_{m+1}) can be added without violating the no-crossing rule. Such graphs seem to have an interesting structure.

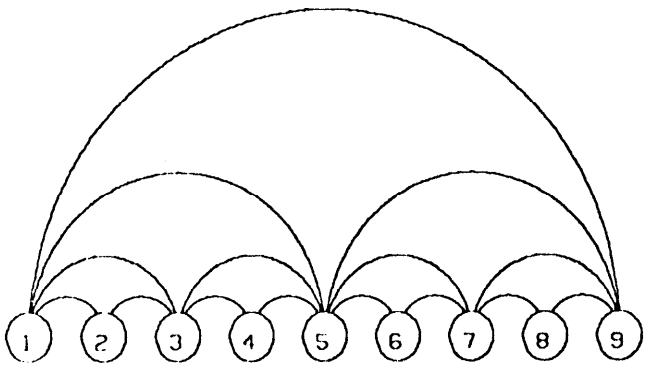
After you have tested your program, it will be run on some test data in a small contest to see whose algorithm finds the smallest rectangles in a reasonable amount of time.

Class notes for November 13.

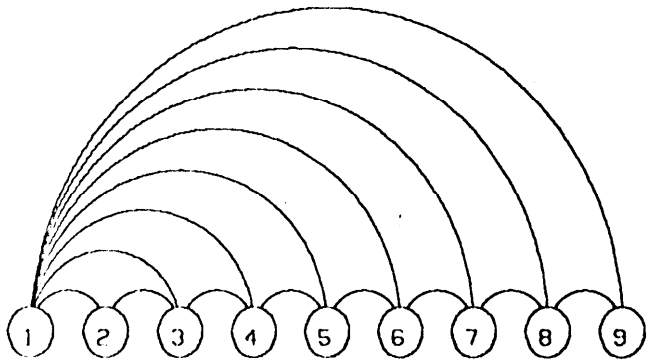
DEK suggested that a good place to start on Problem 5 might be to figure out what the maximum number of edges in an outerplanar graph can be. RLH suggested that you can connect all adjacent vertices, then all those separated by two, then all those separated by four, and so on. The number of edges in a graph with n vertices connected this way is

$$\sum_{0 \leq k \leq \log_2 n} \left\lfloor \frac{n-1}{2^k} \right\rfloor$$

which is just $2n - 3$ whenever n is one more than a power of two. For example the graph with 9 vertices

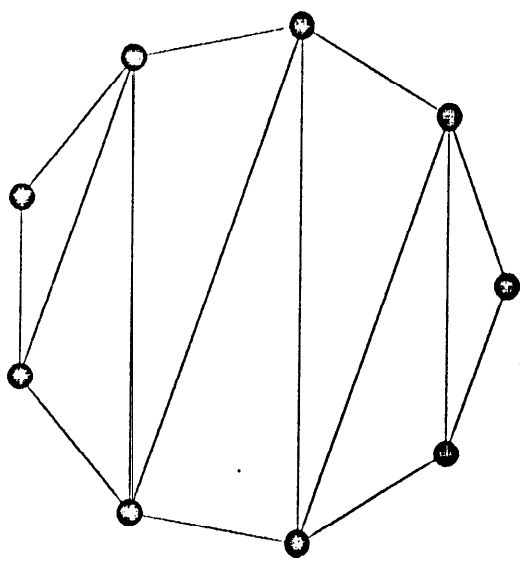


has 15 edges. DD suggested that another construction is to connect all adjacent vertices, then connect the first vertex to the second, third, and so on. This yields $2n - 3$ edges for any graph; for example, the g-vertex graph looks like



The problem with both of these constructions is that neither one satisfies the constraint given in the problem statement that each vertex be of degree four or less.

At this point, DEK observed that if all edges are present between adjacent vertices, and an edge between the first and last vertices, the outerplanar graph can be represented as an n-gon with non-intersecting chords forming the rest of the edges in the graph. The maximum number of edges occurs when the polygon has been divided into triangles, and we get $2n - 3$ edges in such a case. FJB then came up with a construction having $2n - 3$ edges with no vertices having degree more than four. For example,



has 9 vertices and 15 edges.

MMS and DEK discussed the fact that outerplanar graphs have a dual representation, namely the tree found by placing a vertex at the center of each cycle and putting a tree edge across every chord. There are a few complications: extra edges must be added to put the graph into its n-gon representation, and to make the tree a binary tree extra edges must be added to fully triangulate the n-gon.

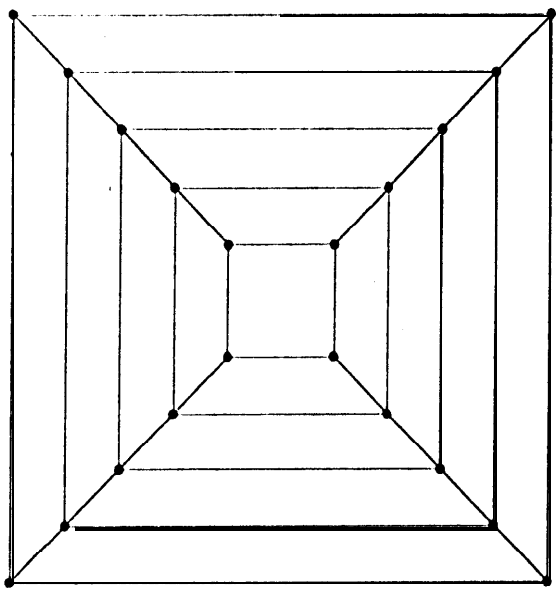
DOH mentioned that in experimenting with some outerplanar graphs, it seemed to him that the vertices of degree four "determined" the layout since they could only have edges attached to them four different ways (each way a rotation of the others), not counting reflections.

DEK suggested that the problem of embedding an outerplanar graph in a rectangular grid might be approached by finding an embedding for some "hardest" graph and then embedding any other graph as a special case. The class finished just as he proposed considering the problem of embedding a binary tree in a rectangular grid.

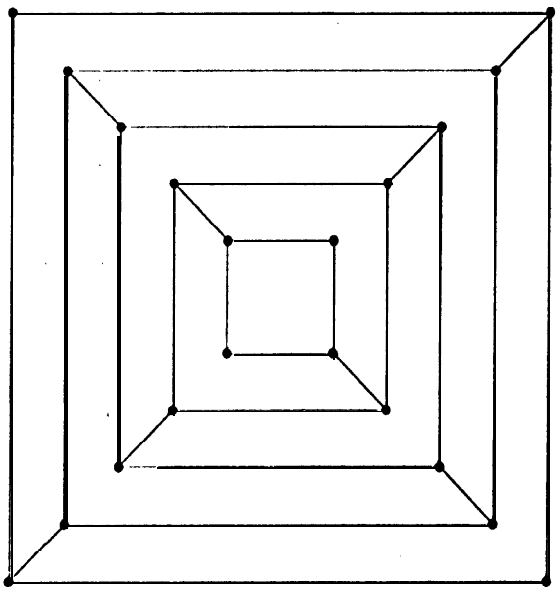
Class notes for November 18.

AAM started the class by announcing that test data describing an outerplanar graph is in CS204.DAT[1,DEK] on SAIL and <CSD.MILLER>PROB5.DAT on SCORE. In addition, five ALTOs will be available for CS204 students to use on a signup basis; each group can sign up for at most two hours a day.

DEK mentioned that he had heard a rumor claiming that outerplanar graphs can be laid out in area linear with respect to the number of vertices in the graph. However, this is not true for all planar graphs, since graph8 such as



require about $16n^2$ area. Even planar graphs with vertices of degree three or less cannot be laid out in linear area, since graphs such as

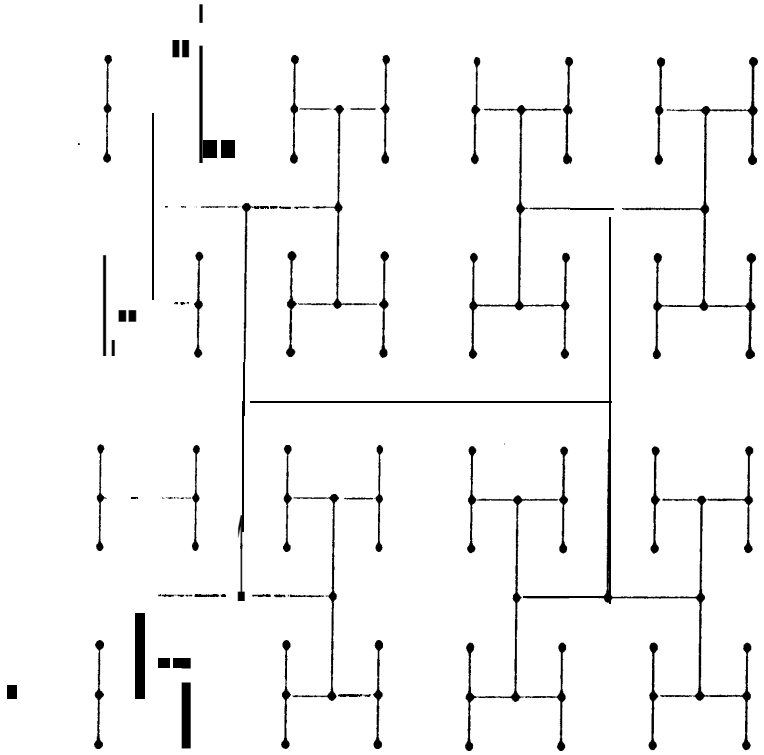


also require about $16n^2$ area. (The difficulty of laying out these particular graphs was first pointed out by Leslie Valiant; an interesting thesis about layout of planar graphs was completed by Don Woods at Stanford in June 1981.)

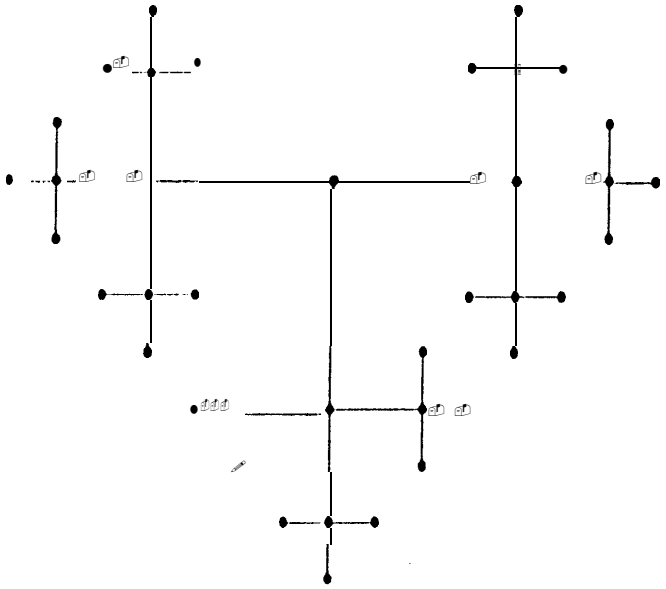
DEK stressed that there are two completely different but equally valid ways to approach Problem 5: one can either try to find a construction that is guaranteed to find an embedding using area linear in the number of vertices (the "theoretical" approach); or one can try to develop heuristics that will find an embedding whose area is very close to the area used by the optimal embedding (the "practical" approach). It would be nice, of course, to find an algorithm that solves both the theoretical and the practical problems simultaneously, but life isn't usually that simple. Theory and practice don't come together closely when really hard problems are involved, at least not in the early stages of research; yet they do support each other.

DEK showed the class a construction for laying out binary trees in linear area, using the so-called "N-method", which seems to be of both theoretical and practical interest. The root of the tree is placed at the center of the construction. Its two descendants are placed to the right and left of it. Their two descendants are placed above and below them. (At this point the construction looks like the letter "H", hence the name.) Subsequent levels are placed in a similar fashion. The full binary

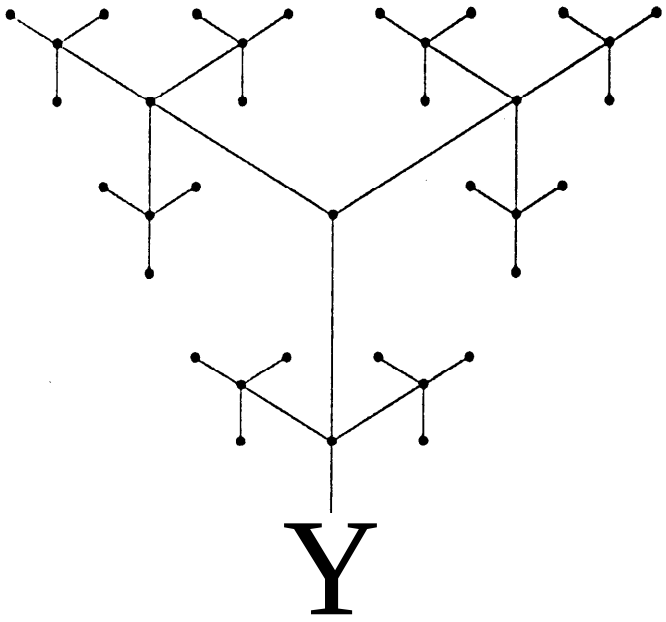
tree with 127 vertices (7 levels) looks like



OP said that since the problem deals with graphs having vertices of degree four, a construction for ternary trees would be needed. DEK tried to draw a "T-method" which looked like **a** fractal:

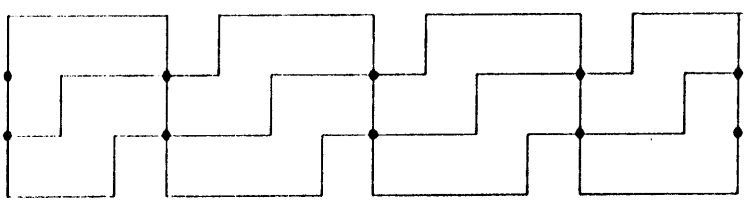


but JDH objected that it didn't look like this construction would use a linear amount of area in the rectangular bounding box. MMS said it might be possible that the wasted space would be some constant fraction of the bounding box, which would preserve the "linear-ness" of the construction, although resulting in a larger constant of linearity. There was some more discussion on just how bad the "T-method" would be, and then OP proposed a "Y-method" resulting in constructions like

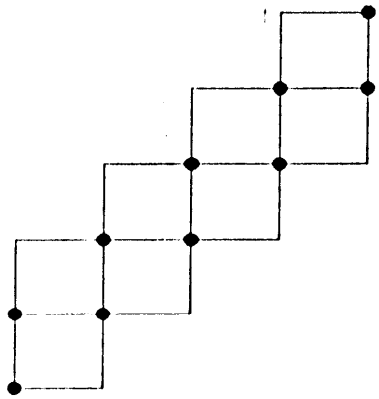


This method seems to require linear area but it is tricky to force it into a rectangular grid. JDH suggested that you might be able to use a slightly modified "H-method" where the four corners of the "H" represented a vertex and its three ancestors. SGD asked why the problem had been restricted to rectangular grids, implying that the use of rectangular geometries in VLSI circuit design might be more a result of available design aids rather than properties of the circuits and materials themselves. DEK replied that he wasn't sure why VLSI is done with rectangular grids, but that when- he had tried using triangular grids for making fonts he had run into problems with different horizontal and vertical resolution.

DEK changed the subject to constructions for laying out triangulated n-gons. SGD suggested a "long, thin" approach, and DD elaborated by drawing an example for a 10-gon:



The class finished as DEK drew a construction for a 10-gon which he pointed out was very efficient in terms of actual area, but relatively inefficient in terms of **area** of the bounding box:



Class notes for November 20.

Today was spent mostly on trying to come up with a method of approaching the problem of embedding an outerplanar graph; no one had come up with a successful approach.

MMS suggested that some kind of "branch-and-bound" method might be applicable if there is some way to tell whether one placement of a vertex in the graph is better than another placement. DEK said that this was using the idea of doing breadth-first search rather than depth-first search. He mentioned that it would probably be infeasible to do a full search on the problem space since most programs of that kind must be limited to search trees with a depth of about ten or fifteen.

DOH suggested that certain impossible layouts could be discarded quickly, especially those involving vertices of degree four. JMM agreed, pointing out that cycles in the graph must be represented as squares or rectangles. JDH said that one method might bc to take the largest cycle in the graph, then find ways to lay out all vertices of degree four in that cycle. DD objected that this method doesn't always yield a layout even though one is possible. He summarized some unpublished work by Leslie Valiant on laying out trees, the basic method being to lay out subtrees in such a way so that there is always a path out of them to connect them together. He admitted, however, that there doesn't seem to be a similar construction for outerplanar graphs. TCH said he thought it would be a good idea to come up with some construction to embed a maximal graph, then just take edges out of it in order to embed any one particular graph.

RSF came up with the idea of using a random method for laying out the graph. Me proposed choosing one vertex, finding the spot where it can be placed with the smallest increase in the bounding box, and repeating this process for each vertex. Following some more discussion, RSF decided that it would be a good idea to 'increase the likelihood of selecting vertices with higher degree. RLH thought

that it would also be better to bias the selection towards vertices connected to already-placed ones. We tried the method on an example graph, and the results weren't optimal but they weren't too bad.

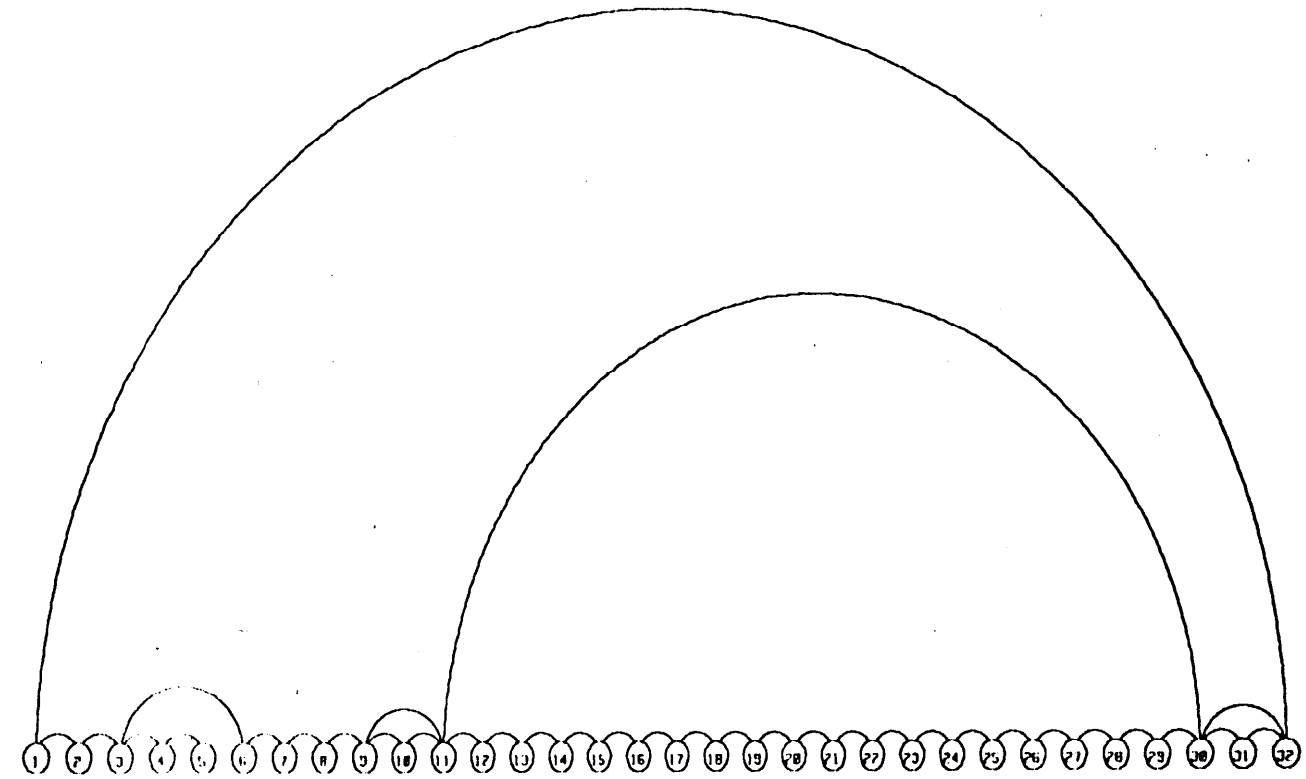
The class finished as DOH mentioned that some small local optimizations could be applied to the final embedding. He gave an example of "closing cycles" by reducing them to the smallest possible rectangles.

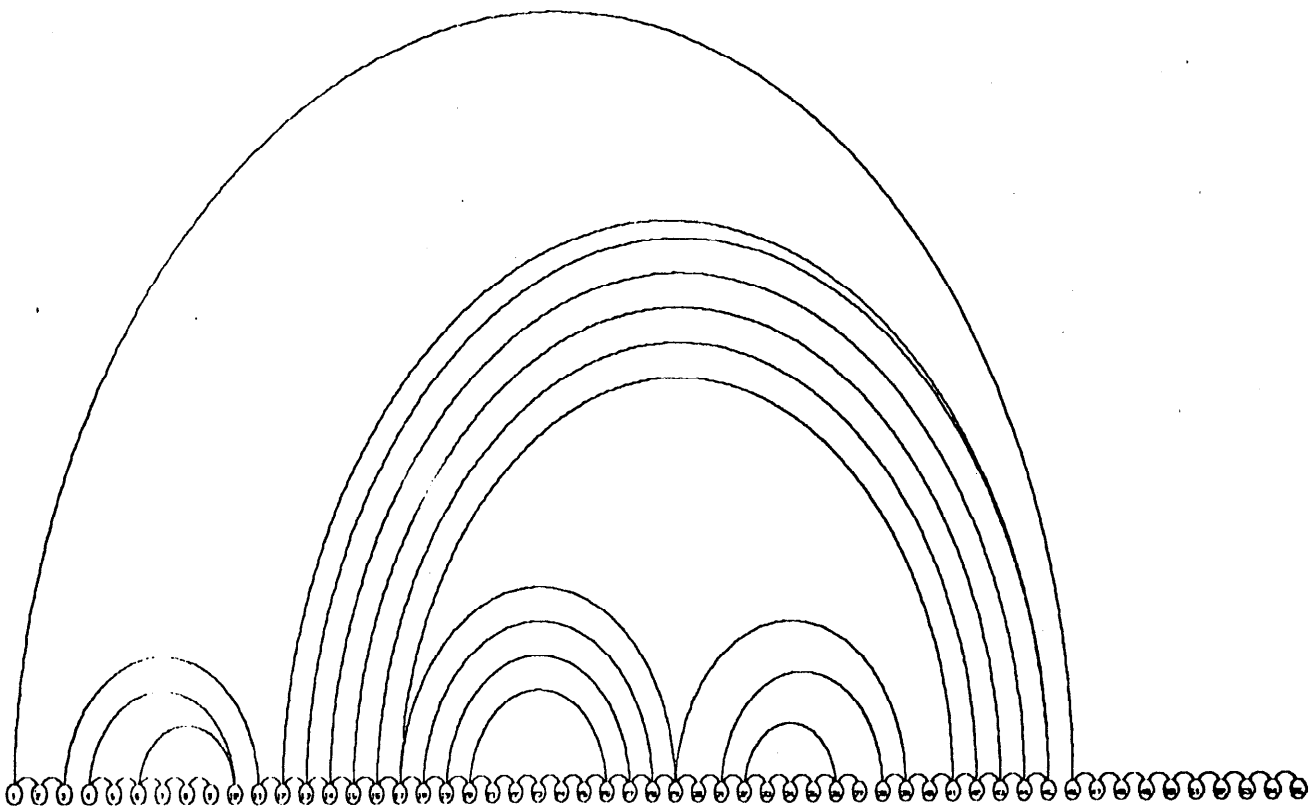
Class notes For **November 25.**

The discussion seemed to be lagging today as no one had any new ideas about the embedding problem. **CXF** mentioned that probably the reason no one had any suggestions was that it is not too clear how to take advantage of the outerplanarity of a graph in order to embed it. DEK brought up the distinction between the "theoretical" and "heuristic" approaches to the problem again. HWT said he **was** trying to figure out a way to lay out the spanning tree of the dual graph in linear area using Valiant's method, then embed the graph around the spanning tree. SGD pointed out that this was no guarantee on coming close to the solution to the problem as stated, since even a 'linear" layout could be far from optimal.

DEK said that he was intrigued by RSF's suggestion to have **a** layout based on a random sequence. One can repeat the random experiment several times and look for common features of the layouts. He mentioned that Monte Carlo methods are often an excellent way to get a good feeling for the problem space. This may have something to do with the fact that "80–20" rules pop up all the time (e.g., 20% of the program takes 80% of the time, 20% of the people own 80% of the wealth, and so on). He also pointed out that it is a common misconception that in "m–n" rules like these it is necessary for $m + n$ to be 100. In fact, the 80-20 rule is the same as the 64-4 rule; the ratio of $\log \frac{m}{100}$ to $\log \frac{n}{100}$ is the significant parameter that determines such a distribution. (See Eq. 6.1-12 in The *Art of Computer Programming.)*

The conversation turned to different ways to instrument and optimize **programs** (from the **"20%** of the code . . .") just as the class ended.





Solutions for problem 5.

Most, people didn't get very far with this problem, although it was apparent from the writeups that they spent a lot of time thinking about it.

RV/JMM/SGD thought that a successful approach to finding a near-optimal layout would involve laying out the degree-four vertices, then working with successively larger cycles in the graph.

OP found an embedding for trees having vertices of degree four in Mandelbrot's book on fractals. He noticed that the 'empty space" in the construction had the same recursive structure as the construction itself, and thought that this might be useful for connecting cycles in an outerplanar graph. However, he was unable to find a way to make these connections.

RSF/DOH implemented the random embedding technique. They limited the runtime of their program by setting a limit on the amount of backtracking done. The program embedded the graph randomly several times, keeping track of the "best" embedding found (the one with the smallest bounding box). In terms of the contest mentioned in the problem statement, they were clearly the winners.

HWT found a construction for embedding outerplanar graphs in linear area. First the graph is drawn as a fully triangulated polygon (extra edges may be added to do this, and some vertices may end up having degree greater than four). Then the dual tree is formed by placing vertices of the dual tree in the center of triangles in the original graph and edges of the dual tree across every interior edge of the original graph. It is easy to see that every vertex in the dual tree has degree three or less. The dual tree is then laid out in linear area using a slight variation of Valiant's method which preserves the topology of the tree (this is important to prevent lines from crossing, since the tree really represents the "space between the original edges"). Then the original graph is laid out around the dual tree, without the edges that were added to triangulate the polygon. By using the fact that the vertices in the original graph had degree four or less, HWT was able to show that this final layout could be done by using a 9×9 grid for each vertex in the dual tree. He indicated that he thought an 8×8 grid would probably actually be large enough, but in any case the layout is linear in the number of vertices in the original graph.

[At this time, DD and HWT are preparing a paper, "On Linear Area Embedding of Planar Graphs", which presents a simplification and extension of HWT's method. If the dual tree is formed by placing one vertex in the center of each triangle (as above) and one vertex outside each exterior edge (so that every edge in the triangulation of the original graph is crossed by exactly one edge of the dual tree), then the dual tree can be laid out and the original graph can be imbedded around it by using at most a 7×7 box for each vertex in the dual tree.

In general this transformation can be applied to any planar graph, but the

dual is a graph rather than a tree. However, if ***k*** is the maximum over all vertices of the shortest path from that vertex to the outermost cycle in the graph, then ***k*** transformations will reduce the planar graph to a binary tree. This tree can be laid out, and each graph can be embedded around the graph it transforms to. (The graphs arc generated, then embedded in the opposite order.) This provides a method for embedding the planar graph in area proportional to ***k.)***

Problem 4. Communication Through Unreliable Links,

Alice wants her computer to send a message to Bill's computer, and she wants to make sure that Bill receives it. Unfortunately, she is in London and Bill **is** in Hong Kong, so she has to use a slow and noisy communications channel named Charlie.

We wish to study the problems associated with this situation, so we will be simulating it using the Alto minicomputers and the Ethernet. You will be writing programs to simulate Alice's and Bill's computers, and Charlie will be simulated by-what could be more unreliable?—a program written by the TA. We will be using a PASCAL-like subset of the MESA language to do the programming for this simulation.

All communication to Charlie will be in the form "send this packet to Alto number n" and communication from Charlie will be in the form "here is a packet from Alto number n". Charlie will always send packets in the order he receives them. However, packets may be delayed for a few dozen seconds, they may **be** lost, or their contents-may be garbled-bits may be changed or lost, or the entire packet may be set to all zeros or all ones. Therefore, packets should contain enough redundancy to give at most about one chance in a billion that a garbled packet is received and considered legitimate.

Alice and Bill must devise a scheme that transmits entire messages (sequences of ten or more packets) and appropriate acknowledgements through Charlie. Note that Alice must be able to retransmit packets that were not received, but Bill's acknowledgement of a received packet might get lost, too! It would be nice if the scheme worked efficiently on those days when Charlie was not losing packets, so simply sending packets one at a time and retransmitting them until an acknowledgement arrives is not a satisfactory scheme. Since Charlie is often slow, **a** new packet should be transmitted before the previous one is acknowledged, in the hope that the previous transmission will take place properly.

Further details about Charlie's interface will be available in time for you to implement your solution to this problem.

Class notes for **December** 2.

AAM started the class by talking a little bit about MESA and handing out some materials to make it easier to use.

DOH said that in the "real world" the biggest problem with communication protocols is making them work with existing protocols built into hardware, since **a** large investment has already been made in those protocols. He also pointed out other important problems that have been finessed out of our Problem 4, for example the question of how to recognize the end of a variable-length message (or **a** partially-transmitted message) on **a** serial line. He observed that in large communication

networks the assumption in Problem 4 that Charlie does not permute the order of ungarblcd messages is not valid, since different messages might travel via different routes. DEK suggested that we try first to solve the problem under the non-permutation assumption, but keep in mind that it would be nice to solve it also without that assumption; then we can compare the efficiency of our best solutions with and without allowable permutations.

JMM mentioned that the nature of the "garbling" on the communication channel must be known before a probability of error less than one part in 10^9 can be guaranteed. DOH said that even without this knowledge, a minimum number of bits of redundancy can be established, namely $\lceil \log_2 10^9 \rceil = 30$. For example, a 16-bit checksum would not be sufficient since such a check applied to randomly garbled data would still come out OK by chance, about one time in every 65536. DEK said that this was what he had in mind when he made up Problem 4, but he realizes now that it isn't correct, since JMM's point is perfectly valid. For example, suppose the nature of the garbling was known to be such that each bit was clobbered with probability p, independent of other bits, where p is some given quantity. Then a 16-bit checksum would be enough redundancy provided p is sufficiently small.

DEK sketched some ideas about message encoding; he said that most encoding schemes arc "linear" in the sense that the message $x_1 \ldots x_n$ is valid if and only if $f(x_1, \ldots, x_n) = c$ where f is some linear function and c is some m-bit constant (here the "message" includes both the data being sent and the redundant checksum). A linear function satisfies $f(\vec{x} \oplus \vec{y}) = f(\vec{x}) \oplus f(\vec{y})$, where $\oplus$ denotes exclusive-or on bit vectors; hence if $\vec{y}$ represents the error vector of clobbered bits, we fail to recognize the error if and only $f(\vec{y}) = 0 \ldots 0$. For example, the ordinary "odd-parity" code is the special case $m = 1$, $c = 1$, $f(x_1, \ldots, x_n) = x_1 \oplus \cdots \oplus x_n$. Any linear function can be rcprcscntcd in the general form $f(x_1, x_2, \ldots, x_n) = x_1 w_1 \oplus x_2 w_2 \oplus \ldots \oplus x_n w_n$, where w_i is the m-bit result of the function f applied to the vector that is all zeros except $x_i = 1$. Thus if the error vector y contains exactly one 1 bit, the error will be dctcctcd provided the w's are all nonzero; if y contains exactly two 1 bits, the error will be detected provided the w's are all distinct.

FNY asked if it would be worth considering error-correcting codes. DEK said such codes arc quite useful in practice, but he would recommend that they not be added to this problem unless it is really easy to do so, since he intended the real focus of Problem 4 to be on the nature of asynchronous parallel processes. If a message is garbled, Alice and Bill can treat it just as if it were totally lost, since lossage is another thing Charlie might do; the message must in general be retransmitted anyway. In other words, error correction would only have the effect of a slight improvement of Charlie's transmission capability; it does not affect the protocols for resending messages and acknowledgments that have been really clobbered.

However, WC talked about error-correction anyway. JMM said that a very simple error-correcting code can be made by putting bits into a rectangular array,

then creating an extra row made up of parity bits for the columns and an extra column made up of parity bits for the rows. If one transmitted bit is changed, we can identify it because its row and column will have bad parity. FNY said this scheme won't work if an error occured in the 'extra" row or column, but it was pointed out that it can be fixed it by adding a parity bit for the entire rectangle. Thus, a message of $n_1 \times n_2$ bits can be encoded by $(n_1 + 1) \times (n_2 + 1)$ bits having even parity in each row and column. This is a simple example of a linear code. For example, if $n_1 = n_2 = 2$, the message bits are xi, x_2, x_3, x_4, and the parity bits are c_1, c_2, c_3, c_4, c_5, then the rectangular array looks like

x_1	x_2	c_5
x_3	x_4	c_4
c_1	c_2	c_3

where $c_1 = x_1 \oplus x_3$, $c_2 = x_2 \oplus x_4$, $c_5 = x_1 \oplus x_2$, $c_4 = x_3 \oplus x_4$, and $c_3 = c_1 \oplus c_2 = c_5 \oplus c_4 = x_1 \oplus x_2 \oplus x_3 \oplus x_4$. The receiver can check the message by checking the parity on, for example, all three rows and the first two columns. In this case $n = 9$, $m = 5$, $c = 00000$, and $f(x_1, x_2, x_3, x_4, c_1, c_2, c_3, c_4, c_5) = (x_1 \oplus x_2 \oplus c_5, x_3 \oplus x_4 \oplus c_4, c_1 \oplus c_2 \oplus c_3, x_1 \oplus x_3 \oplus c_1, x_2 \oplus x_4 \oplus c_2)$. With this assignment of f, we find

$$w_1 = 10010$$
$$w_2 = 10001$$
$$w_3 = 01010$$
$$w_4 = 01001$$
$$w_5 = 00110$$
$$w_6 = 00101$$
$$w_7 = 00100$$
$$w_8 = 01000$$
$$w_9 = 10000.$$

In general, whenever the w_i are distinct and nonzero, it is obvious that we can not only detect single and double errors, we can correct single errors (although we may not be able to distinguish the case of correctable single errors from uncorrectable double errors). This particular rectangular code not only makes the error correction possible, it makes the correction simple.

DEK spoke briefly about Hamming codes for error correction; this is essentially the extreme case where we choose the w's to be all possible distinct nonzero m-bit vectors. The Hamming code scheme uses m bits of redundancy to correct single errors in $2^m - m - 1$ bits of data (unlike the rectangle scheme, which uses m bits of

redundancy to correct single errors in only $(\frac{m-1}{2})^2$ bits of data). To use Hamming codes, m linearly independent m-bit vectors are chosen, and the other nonzero vectors arc given indices from 1 to $2^m - m - 1$ in any convenient way. Then, for each bit x_i, a checksum is started at zero and if $x_i = 1$ the vector with index i is exclusive-or'cd into the checksum. The redundancy bits are computed so that the checksum has a desired value c. At the receiving end, if the computed checksum is different from c, exclusive-or with c will tell which bit is in error (including the case if the error bit is in the redundant bits).

Finally, DEK returned to JMM's original question, which still hadn't been answered, admitting that the problem specification wasn't clear enough. He decided to substitute the following criteria for transmission: (a) Your encoding scheme should be such that it will detect with certainty whenever Charlie alters exactly one or two bits of the message, or when he sets an entire packet to all O's or all 1's. (b) Your scheme should also be such that it will detect with probability at least .999999999 when Charlie has replaced a packet by a packet of completely random bits.

DEK said that we should be ready to talk about lost acknowledgments from Bill to Alice at the next class session.

Class notes for Dcccmbcr 4.

Today DEK led off with the question of whether or not the problem of communicating is solvable, even if Alice is only trying to send one packet. JMM said he didn't think it is, unless some assumptions are made about the reliability and speed of transmission, due to the following problem: when Alice sends Bill a message, in order for Alice to know that Bill has received it Bill must acknowledge it. However, in order for Bill to know that Alice knows that Bill has received the original message, Alice must acknowledge the acknowledgement. But for Alice to know that Bill knows that Alice knows that Bill received the message, . . . the acknowledgements go on forever. Whoever last sent some data will be uncertain whether the data was received, and if their uncertainty is unimportant there is no point in sending the data in the first place. DEK suggested that, at best, the problem must be resolved with one party still active and ready to receive subsequent communications. For example, suppose Bill remains active after sending the first acknowledgement. If Alice doesn't hear from him, she can transmit the message again, while if she does get confirmation she can "go to sleep" and effectively forget about the entire transaction; Bill, on the other hand, must remain on call. Somebody pointed out that they can both "go to sleep" if there is a central server mechanism (like Charlie) that stays active In a sense, the problem of termination has been transformed into the problem of getting started.

For our purposes, we will henceforth ignore the problem of deciding when the last message has been received, concentrating instead on the problem of getting the

other messages across the line.

DEK handed out an article on communication protocols by Stein Krogdahl [R/T 18 (1978), 436–448]. He mentioned that this paper is noteworthy because of the techniques it uses to prove the validity of parallel programs via invariant relations that remain true about the system as a whole.

DEIC illustrated the method treated by Krogdahl in the simplest case by paraphrasing it as follows: Alice keeps a counter A representing how many packets she knows Bill has received; Bill keeps a counter B representing how many packets he knows he has received. Initially A = B = 0; the packets to be sent are numbered consecutively starting with 0. The basic operations performed by Alice and Bill, in some order, are these: (Al) Alice sends packet number A. (A2) Alice receives an acknowledgement 'b' and sets A $\leftarrow b$. (B1) Bill sends an acknowledgement 'B'. (B2) Bill receives packet number j for some j; if B = j he stores it and increases the value of B. Packets and acknowledgements might "disappear" after they are sent; garbled transmissions are treated as equivalent to messages that never arrive.

Now it is possible to prove that, regardless of what order the operations (Al), (A2), (B1), (B2) arc intermixed, the values of A and B will satisfy $B = A$ or $B = A + 1$. Furthermore, the value of b received by Alice in (A2) will always equal either A or A + 1, and the value of j received by Bill in (B2) will always equal either B or $B - 1$. Therefore it suffices for Bill to transmit only the one-bit value 'b mod 2' instead of 'b', and it suffices for Alice to transmit only the one-bit value 'j mod 2' to identify packet number 'j'. In other words, only two kinds of acknowledgements are necessary, and only one bit of a packet is needed to identify it with respect to other packets that have been retransmitted or transmitted before Bill is ready to receive them.

Of course, this scheme isn't efficient enough for our purposes. It can 'be generalized in several ways. DEK stated that if operation (Al) is extended so that Alice sends any of the packets numbered A through $A + k - 1$ inclusive, it can be shown that $A \leq B < A + k$ and that the value of j received by Bill in (B2) will lie in the range $B - k \leq j < B + k$. In such a case the modulus 2 can be replaced by $k + 3$. Furthermore we can generalize (B2) so that Bill will store packet number j whenever $B \leq j < B + I$, where l is some number representing the size of Bill's buffer storage (as k represents the size of Alice's); in this case, Bill is allowed to increase B after (B2) to any value such that all packets whose number is less than B have been stored. It turns out that it now suffices to transmit the value of j mod m_1 with each packet and the value of b mod m_2 with each acknowledgement, whenever $m_1 \geq k + l$ and $m_2 > k + 1$. Verification of these facts is a good exercise in understanding parallel processes.

The above statements are correct only if transmissions are strictly first-in-first-out, however. [After class, DEK decided to look at the method under the assumption that no packet is "passed" by more than q_1 packets in the message queue and no

acknowledgement is "passed" by more than q_2 packets in the acknowledgement queue. In this general case the conditions $m_1 > k + l + q_2$ and $m_2 \geq kq_1 + k + 1$ are necessary and sufficient.]

Class notes for December 9.

Since today was the last official class period (Thursday is reserved for the famous "ask Don Knuth any question"), people were mostly working on finishing up problems four and five.

RSE' and DOH said that they had implemented the random layout method, and had found that one of the major problems with it is a kind of "hill-climbing problem" where it lays out long cycles in a straight line since adding each point increases the bounding box by the smallest amount. Then when the cycle is finally closed a long line must be added alongside the sequence of vertices.

SGD said that he thought we were cheating by using one packet size in the communication problem, since it is possible to take advantage of very long acknowledgements that might not be available in a realistic situation. DEK said that he was preparing a paper based on an optimized form of Krogdahl's algorithm; one that uses a restricted range of integers for the packet identifier rather than any integer. He said it was interesting that the easiest way he had found to prove the algorithm correct is to prove it for unrestricted packet identifiers, then show that the restriction does not affect the arguments in the proof. He likened the process to optimizing a working program. [This paper, "Verification of low-level protocols," was later published in *BIT* (1981).]

The rest of the period was used by HWT to describe a method he devised to lay out outcrplanar graphs in linear area. Since his method is described in detail in the writeup for problem five, it will not be discussed here.

MESA programs prepared for use in Problem 4

```
-- "Charles" program.  CS 204, Fall 1988, Knuth.
-- This program serves as a noisy communications interface between
-- "Alice" and "Bill" using the Pup protocol on the Xerox ethernet.
-- written by Allan Miller

DIRECTORY
        IODefs: FROM "iodefs" USING [
                WriteString,
                WriteDecimal,
                WriteLine],
        ABCDefs: FROM "abcdefs" USING [
                AliceBillSocket,
                CharlieSocket,
                NumBuffers,
                Buffet-Size,
                PacketLen,
                GroupNumberType,
                ABCPacket],
        RandomDefs: FROM "randomdefs" USING [
                Random],
        GarbleDefs: FROM "garbledefs' USING [
                LosePacket,
                Garble],
        PupTypes: FROM "puptypes' USING [
                PupNetID,
                PupHostID],
        PupDefs: FROM "pupdefs" USING [
                PupSocket,
                PupBuffer,
                PupAddress,
                SecondsToTocks,
                AdjustBufferParms,
                PupPackageMake,
                PupSocketMake,
                SetPupContentsWords,
                GetFreePupBuffer,
                ReturnFreePupBuffer];

Charlie: PROGRAM IMPORTS IODefs, PupDefs, GarbleDefs, RandomDefs =

BEGIN
OPEN IODefs, PupDefs, PupTypes, ABCDefs, GarbleDefs, RandomDefs;

Socket: PupSocket; -- all pup io goes through here
Buffer: PupBuffer; -- data goes here
Packet: ABCPacket; -- info goes in here
Gnum:  GroupNumberType; -- group that sent an incoming packet
Contact: TYPE = RECORD [
        Count: INTEGER, -- how many people have made contact
        TotalDelay: INTEGER, -- how much delay this group has built up
        AliceAddr: PupAddress, -- note that Charlie doesn't really care
        BillAddr: PupAddress]; -- which one is Alice and which one is Bill
Table: ARRAY GroupNumberType OF Contact;
MaxDelay: INTEGER = 15; -- largest amount of total delay
LastInQueue: CARDINAL = NumBuffers - 4; -- number of packets that Charlie can queue
                                        -- Pup package uses 2 of total, ethernet
                                        -- driver also uses 2
QueueObject: TYPE = RECORD [
        Delay: INTEGER,
        Contents: PupBuffer];
```

```
PacketQueue: ARRAY [1..LastInQueue] OF QueueObject;
EndOfQueue:  CARDINAL;

AcknowledgeContact: PROCEDURE [who: PupAddress]=
BEGIN
b: PupBuffer;
    b _ GetFreePupBuffer[];
    b↑.source _ Socket↑.getLocalAddress[];
    b↑.dest _ who;
    SetPupContentsWords[b, PacketLen];
    Packet _ LOOPHOLE[@b↑.pupBody];
    Packet↑.GroupNumber _ 0; -- Group 0 is Charlie himself
    Packet↑.MessageType _ ConfirmContact;
    Socket↑.setRemoteAddress[who];
    Socket↑.put[b] -- note that this calls ReturnFreePupBuffer
END;

QueueForSending: PROCEDURE [g: GroupNumberType, b: PupBuffer] =
BEGIN
 DoIt: PROCEDURE[a: PupAddress] =
        BEGIN
        P: ABCPacket;
                WriteString["Queueing packet for group "];
                WriteDecimal[g];
                WriteLine["."];
                b↑.source _ Socket↑.getLocalAddress[];
                b↑.dest _ a;
                SetPupContentsWords[Buffer, PacketLen];
                EndOfQueue _ EndOfQueue + 1;
                IF EndOfQueue > LastInQueue THEN
                        BEGIN
                                WriteLine["Losing packet for group "];
                                WriteDecimal[g];
                                WriteLine[" due to lack of buffer space!! !"];
                                ReturnFreePupBuffer[b];
                                EndOfQueue _ LastInQueue;
                        END
                ELSE
                        BEGIN
                                WriteString["Packet for group "];
                                WriteDecimal[g];  WriteString[":  "];
                                IF LosePacket[] THEN
                                        BEGIN
                                                WriteLine["lost"];
                                                ReturnFreePupBuffer[b];
                                                EndOfQueue _ EndOfQueue - 1;
                                        END
                                ELSE
                                        BEGIN
                                        SecondsToDelay: INTEGER;
                                                P _ LOOPHOLE[@b↑.pupBody];
                                                WriteLine[Garble[DESCRIPTOR[P↑.Message]]];
                                                PacketQueue[EndOfQueue].Contents _ b;
-- The total amount of delay for a group is kept under MaxDelay to prevent them from
-- flooding us with packets.  The "+1" in the 3rd line is because this is the number of
-- checks before sending...
                                                SecondsToDelay _ Random[MaxDelay+1-Table[g]
.TotalDelay];
                                                Table[g].TotalDelay _ Table[g ].TotalDelay +
 SecondsToDelay;
                                                PacketQueue[EndOfQueue].Delay _ Table[g].To
talDelay + 1;
                                        END;
```

```
                        END;
        END;

    SELECT bt.source FROM
        Table[g].AliceAddr => DoIt[Table[g].BillAddr];
        Table[g].BillAddr =>  DoIt[Table[g].AliceAddr];
        ENDCASE =>
            BEGIN
                WriteString["Someone used the group number "];
                WriteDecimal[g];
                WriteLine[" when they shouldn't have!! !"]
            END
END;

SendPendingPackets: PROCEDURE =
BEGIN
Pnum, I: CARDINAL;
        Pnum  1 ;
        WHILE-Pnum <= EndOfQueue DO
                PacketQueue[Pnum].Delay _ PacketQueue[Pnum].Delay  - 1;
                IF PacketQueue[Pnum].Delay <= 8 THEN
                        BEGIN
                                WriteLine["Sending a packet."];
                                Socket↑.setRemoteAddress[PacketQueue[Pnum].Contents↑.dest];
                                Socket↑.put[PacketQueue[Pnum].Contents]; --this calls Retur
nFreePupBuffer
                                FOR I IN [Pnum..EndOfQueue) DO
                                        PacketQueue[ I] _ PacketQueue[ I+1];
                                ENDLOOP;
                                EndOfQueue _ EndOfQueue  - 1;
                                Pnum _ Pnum - 1; --  granted, this is a little hackish
                        END;
                Pnum _ Pnum + 1;
    ENDLOOP;
END;

--  initialize
EndOfQueue_ 0;
FOR Gnum IN GroupNumberType DO
        Table[Gnum].Count _8 ;
        Table[Gnum].TotalDelay _ 8;
ENDLOOP;
AdjustBufferParms[NumBuffers, BufferSize];
PupPackageMake[ ];
Socket _  PupSocketMake[CharlieSocket,
                [PupNetID[0], PupHostID[0], AliceBillSocket], -- this addr never really use
d
                SecondsToTocks[ 1]]; -- check the queue every second

WHILE TRUE DO
    SendPendingPackets[ ];
    -- Get a pup
    Buffer  _ Socket↑.get[ ];
    WHILE Buffer = NIL DO
        SendPendingPackets[];
        Buffer _ Socket↑.get[ ]; -- no filtering is done, so Charlie gets
                                 -- everything that's addressed to him
        ENDLOOP;
    -- Do the right thing with it
    Packet _ LOOPHOLE[@Buffer↑.pupBody];
    Gnum _ Packet↑.GroupNumber;
    WriteString["Received a packet from group "];
        WriteDecimal[Gnum];
```

```
        WriteString[", which has count "];
        WriteDecimal[Table[Gnum].Count];
        WriteLine["."];
        SELECT Packett.MessageType FROM

        EstablishContact =>
            BEGIN
                SELECT Table[Gnum].Count FROM
                    8 => --this is the first connection
                        BEGIN
                            Table[Gnum].AliceAddr _ Buffert.source;
                            Table[Gnum].Count _ 1
                        END;
                    1 => --this is the second connection; acknowledge both
                        BEGIN
                                -- fix bug where alice gets started twice before bill
                                IF Buffert.source # Table[Gnum].AliceAddr THEN
                                        BEGIN
                                            Table[Gnum].BillAddr _ Buffert.source;
                                            Table[Gnum].Count _ 2;
                                            WriteString["Establishing contact for group "];
                                            WriteDecimal[Gnum];
                                            WriteLine["."];
                                            AcknowledgeContact[Table[Gnum].AliceAddr];
                                            AcknowledgeContact[Table[Gnum].BillAddr];
                                        END;
                        END;
                    2 => --too many connections, try to recover
                        SELECT Buffert.source FROM
                            Table[Gnum].AliceAddr, Table[Gnum].BillAddr =>
                            -- someone restarted their program; let it slide
                                AcknowledgeContact[Buffert.source];
                            ENDCASE =>
                            -- someone moved or made a mistake; too bad for them
                                BEGIN
                                    WriteString["Too many contacts for group "];
                                    WriteDecimal[Gnum];
                                    WriteLine["!!!"]
                                END;
                    ENDCASE => --this can never happen
                        WriteLine["Dryrot--bad value of count!! !"];
                ReturnFreePupBuffer[Buffer]
            END;

        SendPacket =>
            IF Table[Gnum].Count < 2 THEN
                BEGIN
                    WriteString["Group "];
                    WriteDecimal[Gnum];
                    WriteLine[" is attempting to communicate with a nonexistent partner!! !"
];
                    ReturnFreePupBuffer[Buffer]
                END
            ELSE QueueForSending[Gnum,Buffer];

        FinishContact =>
            BEGIN
                IF Table[Gnum].Count < 1 THEN
                    BEGIN
                        WriteString["Group "];
                        WriteDecimal[Gnum];
                        WriteLine[" is doing too many CloseContacts!! !"]
                    END
```

```
                ELSE
                    Table[Gnum].Count _ Table[Gnum].Count - 1;
                ReturnFreePupBuffer[Buffer]
            END;

        ENDCASE =>
            WriteLine["Someone sent me a bad packet type!!!"];
  ENDLOOP; --WHILE TRUE
END.
```

```
RandomDefs: DEFINITIONS =
BEGIN

InitializeRandom: PROCEDURE [Seed: CARDINAL];
GeneratorOut: PROCEDURE RETURNS [CARDINAL];
Random: PROCEDURE [MaxVal: CARDINAL] RETURNS [CARDINAL];

END.
```

```
-- Random number generator. CS 204, Fall 1988, Knuth.
-- 'This is used by Charlie in his garbling of messages. From 2.3.6. of the Art of Computer
 Programming.
-- written by Allan Miller

DIRECTORY
        RandomDefs: FROM "randomdefs";

RandomSubs: PROGRAM EXPORTS RandomDefs =
BEGIN

-- parameters for subtractive random number generator
j: CARDINAL ▫ 24; -- j and k from table 3.2.2-1, with k > 58
k: CARDINAL = 55;
c1: CARDJNAL = k-j;
c2: CARDINAL = j+1;
d: CARDINAL = 21; -- d should be about 8.382 k
BigNum: CARDINAL = 580;
StartSeed: CARDINAL = 314;
Initialized: BOOLEAN FALSE; -- tells whether or not InitializeRandom has been called
NextNum: CARDINAL; ---tells which number in array is next random one
Rands: ARRAY [1..k] OF CARDINAL;

InitializeRandom: PUBLIC PROCEDURE [Seed: CARDINAL] =
BEGIN
ii, iii: JNTEGER;
jj, kk: INTEGER;
jj _ Seed MOD BigNum;
Rands[k] _ jj;
kk _ 1;
FOR ii IN[1..k) DO
        iii _ (d*ii) MOD k;
        Rands[iii] _ kk;
        kk _ jj - kk;
        IF kk < 8 THEN kk _ kk + BigNum;
```

```
        jj _ Rands[iii];
ENDLOOP;
-- "warm up" the generator
RecomputeRandom[];
RecomputeRandom[];
RecomputeRandom[];
NextNum _ 0;
Initialized _ TRUE;
END;

RecomputeRandom: PROCEDURE =
BEGIN
ii: INTEGER;
jj: INTEGER;
FOR ii IN [1..j] DO
        jj _ Rands[ii] - Rands[ii+c1];
        IF jj < 0 THEN jj _ jj + BigNum;
        Rands[ii] _ jj;
ENDLOOP;
FOR ii IN [c2..k] DO'
        jj _ Rands[ii] - Rands[ii-j];
        IF jj < 0 THEN jj _ jj + BigNum;
        Rands[ii] _ jj
ENDLOOP;
END;

GeneratorOut: PUBLIC PROCEDURE RETURNS [CARDINAL] =
BEGIN
NextNum _ (NextNum MOD k) + 1;
IF NextNum = 1 THEN RecomputeRandom[];
RETURN[Rands[NextNum]];
END;

Random: PUBLIC PROCEDURE [MaxVal: CARDINAL] RETURNS [CARDINAL] =
BEGIN
IF NOT Initialized THEN InitializeRandom[StartSeed];
RETURN[(MaxVal*GeneratorOut[])/BigNum];
END;

END.
```

```
DIRECTORY
        ABCDefs: FROM "abcdefs" USING [
                PacketBody];

GarbleDefs: DEFINITIONS =
BEGIN
OPEN ABCDefs;

LosePacket: PROCEDURE RETURNS [BOOLEAN];
Garble: PROCEDURE [p: PacketBody] RETURNS [STRING);
END.
```

```
DIRECTORY
        GarbleDefs: FROM "garbledefs",
        ABCDefs: FROM "abcdefs" USING [
                MesgLen,
                PacketBody],
        MiscDefs: FROM "miscdefs" USING [
                CurrentTime],
        InlineDefs: FROM "inlinedefs" USING [
                LowHalf,
                BITNOT,
                BITSHIFT,
                BITXOR],
        RandomDefs: FROM "randomdefs" USING [
                InitializeRandom,
                Random];

GarbleSubs: PROGRAM
        IMPORTS RandomDefs, InlineDefs, MiscDefs
        EXPORTS GarbleDefs =
BEGIN
OPEN ABCDefs, RandomDefs, InlineDefs, MiscDefs;

Startup: BOOLEAN _ TRUE;

LosePercentage: CARDINAL = 50; -- about as reliable as Allan himself
SwapPercentage: CARDINAL = 10; -- switching two words
BitZapPercentage: CARDINAL = 10; -- changing one bit
TwoBitZapPercentage: CARDINAL = 10; -- changing two bits
OnesPercentage: CARDINAL = 10; -- setting to all ones
ZerosPercentage: CARDINAL = 10; -- setting to all zeros
GarblePercentage: CARDINAL = SwapPercentage + BitZapPercentage +
                                TwoBitZapPercentage + OnesPercentage + ZerosPercentage;

LosePacket: PUBLIC PROCEDURE RETURNS [BOOLEAN] =
BEGIN
        IF Startup THEN -- I'm taking advantage of the fact that LosePacket is called first
                BEGIN
                        InitializeRandom[LowHalf[CurrentTime[]]];
                        Startup _ FALSE;
                END;
        RETURN[Random[100] < LosePercentage];
END;

Garble: PUBLIC PROCEDURE [p: PacketBody] RETURNS [STRING] =
BEGIN
g: CARDINAL; -- used to figure out how to garble the message
-- this code decides whether or not to garble, then splits garbles up as desired
IF Random[100] >= GarblePercentage THEN RETURN["ungarbled"];
g _ Random[GarblePercentage];

IF g < SwapPercentage THEN
        BEGIN
                i,j: CARDINAL;    t: UNSPECIFIED;
                i   Random[MesgLen]; j   Random[ MesgLen];
                ---prevent the obvious bug
                WHILE i = j D O j _ Random[MesgLen]; ENDLOOP;
                t _ p[i]; p[i] _ p[j]; p[j]_ t;
                RETURN["swapped two words"];
        END;
g _ g - SwapPercentage;

IF g < BitZapPercentage THEN
        BEGIN
```

```
                w: CARDINAL;  -- word number to zap
                w _ Random[MesgLen];
                p[w] _ BITXOR[p[w], BITSHIFT[1, Random[16]]];
                RETURN["zapped one bit'];
        END;
g _ g - BitZapPercentage;

IF g < TwoBitZapPercentage THEN
        BEGIN
                wl,w2: CARDINAL;  -- words to zap (note that bits in different words are zap
ped)
                wl _ Random[MesgLen]; w2 _ Random[MesgLen];
                WHILE wl = w2 DO w2 _ Random[MesgLen]; ENDLOOP;
                p[wl] _ BITXOR[p[wl], BITSHIFT[ 1, Random[16]]];
                p[w2] _ BITXOR[p[w2], BITSHIFT[ 1, Random[16]]];
                RETURN["zapped two bits"];
        END;
g _ g -  TwoBitZapPercentage;

IF g < OnesPercentage THEN
        BEGIN
                i: CARDINAL;
                FOR i IN [0..MesgLen) DO p[i] _ BITNOT[0]; ENDLOOP;
                RETURN["set to all ones"];
        END;
g _ g - OnesPercentage;

IF g < ZerosPercentage THEN
        BEGIN
                i: CARDINAL;
                FOR i IN [0..MesgLen) DO p[i] , 0; ENDLOOP;
                RETURN["set to all zeros"];
        END;

RETURN["Hey! ! !  There's a bug in the Garble routine!!!"];
END;

END.
```

```
DIRECTORY
        PupDefs: FROM "pupdefs" USING [
                PupSocketID];

ABCDefs: DEFINITIONS =
BEGIN
OPEN PupDefs;

AliceBillSocket: PupSocketID =[0,45B];
CharlieSocket: PupSocketID = [0,46B];
DelayInSeconds: CARDINAL = 0;
MaxGroup: CARDINAL = 15; -- largest number of groups working on this problem
GroupNumberType: TYPE = [0..MaxGroup); -- suitable for arrays GROUP 0 IS CHARLIE!!!
NumBuffers: CARDINAL = 30;  -- number of pup buffers to allocate
BufferSize: CARDINAL = 70;  -- this only needs to be this big in case the
                                -- program is ever run at PARC (network info, etc.)
MesgLen: CARDINAL = 20; -- length of packet IF LARGE, EXPAND BufferSize!!!
PacketLen: CARDINAL = MesgLen + 2; -- length that actually gets sent
```

```
ABCMesgType: TYPE = {EstablishContact, SendPacket, ConfirmContact, FinishContact}
ABCPacketObject: TYPE = RECORD [
        GroupNumber: CARDINAL, --ID of group that sent packet
        MessageType: ABCMesgType, --what kind of message it is
        Message: ARRAY [0..MesgLen) OF UNSPECIFIED
        ];
ABCPacket: TYPE = POINTER TO ABCPacketObject;
PacketBody: TYPE = DESCRIPTOR FOR ARRAY [0..MesgLen) OF UNSPECIFIED;

-- stuff from ABCSubs
SetUpContact: PROCEDURE [g: GroupNumberType];
SendPacket: PROCEDURE [p: PacketBody];
ReceivePacket: PROCEDURE [p: PacketBody] RETURNS [BOOLEAN];
CloseContact: PROCEDURE;
END.
```

```
-- Procedures used by Alice and Bill to communicate with Charlie.
-- written by Allan Miller

DIRECTORY
        IODefs: FROM "iodefs" USING[
                WriteDecimal,
                WriteString,
                WriteLine],
        ABCDefs: FROM "abcdefs" USING [
                NumBuffers,
                BufferSize,
                AliceBillSocket,
                CharlieSocket,
                GroupNumberType,
                PacketBody,
                PacketLen,
                MesgLen,
                ABCPacket],
        PupDefs: FROM "pupdefs" USING [
                AdjustBufferParms,
                PupPackageMake,
                PupPackageDestroy,
                PupSocketMake,
                PupSocketDestroy,
                PupSocket,
                SetPupContentsWords,
                PupBuffer,
                PupAddress,
                ReturnFreePupBuffer,
                GetFreePupBuffer,
                Tocks],
        Pup-Types: FROM "puptypes" USING [
                allNets,
                allHosts];

ABCSubs: PROGRAM
        IMPORTS ABCDefs, PupDefs, IODefs
        EXPORTS ABCDefs =
BEGIN
OPEN ABCDefs,PupDefs,PupTypes, IODefs;
```

```
Socket: PupSocket;
Buffer: PupBuffer;
Packet: ABCPacket;
OurGroup: GroupNumberType; -- to save for later use
CharlieAddr: PupAddress; -- ditto

SetUpContact: PUBLIC PROCEDURE [g: GroupNumberType] =
BEGIN
    OurGroup g ;
    -- Set up-the pup package
    AdjustBufferParms[NumBuffers, BufferSize];
    PupPackageMake[ ];
    Socket _ PupSocketMake[AliceBillSocket,
                [allNets, allHosts, CharlieSocket],
                Tocks[1]]; -- very short delay in this one, note that 0 does NOT work!
    -- Send Charlie our group number
    Buffer _ GetFreePupBuffer[ ];
    Buffer↑.source _ Socket↑.getLocalAddress[ ];
    Buffer↑.dest _ [allNets, allHosts, CharlieSocket];
    SetPupContentsWords[Buffer,PacketLen]; -- size of packet
    Packet _ LOOPHOLE[@Buffer↑.pupBody];
    Packet↑.GroupNumber _ OurGroup;
    Packet↑.MessageType _ EstablishContact;
    Socket↑.put[Buffer]; -- this also does a ReturnFreePupBuffer
    -- and wait for his reply
    Buffer _ Socket↑.get[ ];
    WHILE Buffer = NIL DO Buffer_ Socket↑.get[ ]; ENDLOOP;
    -- take the socket out of broadcast mode (Charlie won't move)
    CharlieAddr _ Buffer↑.source;
    Socket↑.setRemoteAddress[CharlieAddr];
    -- better safe than sorry...
    Packet _ LOOPHOLE[@Buffer↑.pupBody];
    IF Packet↑.MessageType # ConfirmContact THEN
        WriteLine["Terrible error!!! Charlie gave a bad confirmation!! !"];
END;

SendPacket: PUBLIC PROCEDURE [p: PacketBody] =
BEGIN
I: [0..MesgLen);
    -- send a packet with the info in p to Charlie
    Buffer _ GetFreePupBuffer[ ];
    Buffer↑.source _ Sockett.getLocalAddress[];
    Buffer↑.dest _ CharlieAddr;
    SetPupContentsWords[Buffer,PacketLen]; -- size of packet
    Packet _ LOOPHOLE[@Buffer↑.pupBody];
    Packet↑.GroupNumber _ Out-Group;
    Packet↑.MessageType _ SendPacket;
    FOR I IN [0..MesgLen) DO Packet↑.Message[I] _ p[ I]; ENDLOOP;
    Sockett.put[Buffer]; -- also does a ReturnFreePupBuffer
END;

ReceivePacket: PUBLIC PROCEDURE [p: PacketBody] RETURNS [BOOLEAN] =
BEGIN
I : [0..MesgLen);
    -- try to receive a packet from Charlie and put the information in p
    Buffer Socket↑.get[ ];
    IF Buff&- = NIL THEN RETURN[FALSE];
    -- try to make sure it's really from Charlie
    IF Buffer↑.source # CharlieAddr THEN
        BEGIN
            WriteLine["Terrible error! Someone is trying to impersonate Charlie!! !"];
            RETURN[FALSE]
        END;
```

```
    -- try to make sure it contains valid data
    Packet _ LOOPHOLE[@Buffer↑.pupBody];
    IF Packet↑.GroupNumber # OurGroup THEN
        BEGIN
            WriteLine["Terrible error!  Charlie sent a packet to the wrong group!! !"];
                WriteString["He sent to group "];
                WriteDecimal[Packet↑.GroupNumber];
                WriteString[" but we were expecting "];
                WriteDecimal[OurGroup];
                WriteLine["!!!"];
            RETURN[FALSE]
        END;
    IF Packet↑.MessageType # SendPacket THEN
        BEGIN
            WriteLine["Terrible error!  Charlie didn't send us data!!!"];
                WriteString["He sent us a packet of type "];
                WriteDecimal[LOOPHOLE[Packet↑.MessageType,INTEGER]];
                WriteString[" but we were looking for type "];
                WriteDecimal[LOOPHOLE[SendPacket,INTEGER]];
                WriteLine["!!!"];
            RETURN[FALSE]
        END;
    -- if we survive to here everything "must" be all right
    FOR I IN [0..MesgLen) DO p[I] _ Packet↑.Message[I]; ENDLOOP;
    ReturnFreePupBuffer[Buffer];
    RETURN[TRUE]
END;

CloseContact: PUBLIC PROCEDURE =
BEGIN
    Buffer _ GetFreePupBuffer[];
    Buffer↑.source _ Socket↑.getLocalAddress[];
    Buffer↑.dest _ CharlieAddr;
    SetPupContentsWords[Buffer,PacketLen]; -- size of packet
    Packet _ LOOPHOLE[@Buffer↑.pupBody];
    Packet↑.GroupNumber _ OurGroup;
    Packet↑.MessageType _ FinishContact;
    Socket↑.put[Buffer]; -- this also does a ReturnFreePupBuffer
    PupSocketDestroy[Socket];
    PupPackageDestroy[];
END;

END.
```

Solutions for problem 4.

The most interesting variety in the solutions for this problem was the number of different schemes people used for error detection. OP/JP used a rather simple scheme of making the message size half the packet size and repeating each message twice in a packet. ML/MMS/JJW computed a 2-word Hamming code for the data in each packet and put it at the end of the packet. DOH/RSF computed a "vertical" parity word in which each bit served as the parity for the corresponding bit in all data words, and several "horizontal" parity words in which each bit served as the parity for one data word. RLH/PB had in interesting scheme where a checksum was formed as $\sum d_i p_i$ where d_i is the i^{th} data word in the packet and p_i is the i^{th} prime number (PB proved that this scheme will detect **all** single-word errors). JDH/FNY used a scheme which was kind of fine-tuned to the errors Charles produced: they made one checksum which was the XOR of all words XOR'ed with 31415 (the 31415 was to catch the all zeros and all ones errors), and a two more checksums y_1 and y_2 which were made by XORing data words into y_1 and y_2, rotating y_1 one bit every operation and y_2 one hit every eight operations. Since the packet size was less than 64, this scheme caught all double-bit errors.

Everyone used a scheme very similar to the one presented in class for a message protocol. Everyone solved the "termination problem" by having Alice send a large number of termination packets to Bill and assuming that at least one got through.

Cast of Characters

Staff

DEK

Don Knuth

AAM

Allan Miller

Students

CXF'

picture not available

Chris Fraley

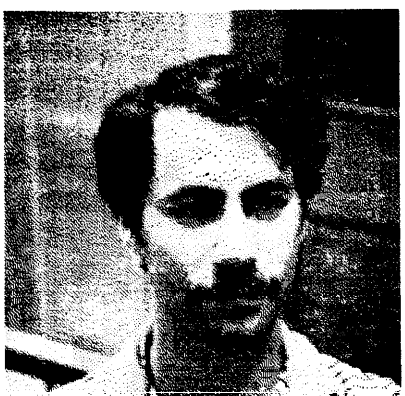
DOH

Doug Hartman

FJB

Duffy Boyle

FNY



Frank Yellin

GMK

Gabriel Kuper

HWT

Howard Trickey

JDH

John Hobby

JJW

Joe Weening

JMM

Jitendra Malik

JP

Jerry Plotnick

ML

Mark Lake

MMS

Mike Spreitzer

OP

Oren Patashnik

PB

Per Bothner

PEV

Paul Vianna

PHW

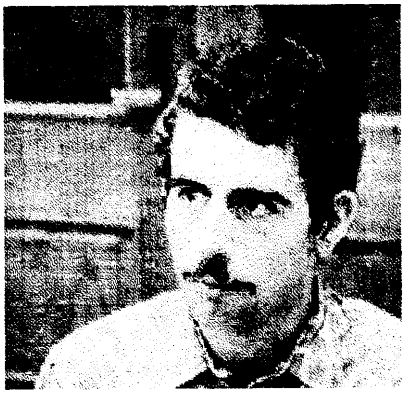
Pat Worley

RLH

picture not available

Bob Hess

RSF



Ross Finlayson

RV

Rick Vistnes

SGD

Stefan Demetrescu

Auditors

DEK2

Dan Kolkowitz

MPH

picture not available

Martin Haeberli

PPH

Peter Hochschild

Visitors

DD

Danny Dolev

JMC

John McCarthy

TCH

picture not available

T. C. Hu