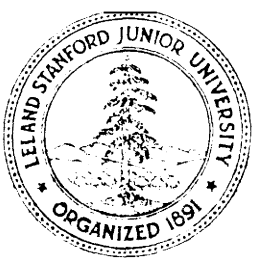
A PROGRAMMING AND PROBLEM-SOLVING SEMINAR

by

Chris Van Wyk and Donald E. Knuth

STAN-CS-79-707
January 1979

COMPUTER SCIENCE DEPARTMENT
School of Humanities and Sciences
STANFORD UNIVERSITY



Computer Science Department
Report No. STAN-CS-79-707

A PROGRAMMING AND PROBLEM-SOLVING SEMINAR

by

Christopher J. Van Wyk* and Donald E. Knuth

This report contains edited transcripts of the discussions held in Stanford's course CS 204, Problem Seminar, during autumn quarter 1978. Since the topics span a large range of ideas in computer science, and since most of the important research paradigms and programming paradigms came up during the discussions, these notes may be of interest to graduate students of computer science at other universities, as well as to their professors and to professional people in the "real world."

*Partially supported by a National Science Foundation Graduate Fellowship.

The production of this report was supported in part by the IBM Corporation.

Table of Contents

Notes from September 28

We spent a large portion of the first class meeting learning one another's names and covering administrative details. DEK took time briefly to explain certain essential facts about the course. First, we're working to solve problems together, not competing to see who can come up with the cleverest ideas or most elegant programs. Moreover, valuable as it is, even this joint problem-solving is not the main focus of the course: it is a vehicle by which we hope to gain greater insight into the important processes of problem-solving and research.

To this end, we'll sometimes step back to look at our work on a problem, trying to identify both the fruitful approaches we tried and the ones that led us nowhere, to explain why they impress us that way, and to see in retrospect what we could have done better.

These notes are intended to record our class discussions about the problems and what we learn about problem-solving through them. Frequently, they will also discuss the solutions turned in for grading, mentioning particularly noteworthy efforts and including a copy of one of the nicer solutions.

Here are questions that will be asked about your solutions:

- Is the program correct?
- Is it nicely structured and well-documented, hence easily understood?
- Are the data structures and algorithms reasonable?
- Did you use features of the SAIL language appropriately, or is there a much better way to accomplish what you did?
- Does your writeup show careful reflection about the work you did leading up to the solution, including results you ultimately discarded?
- Did you use features of the English language appropriately, or is there a much better way to say what you said?

Observe that the writeups are *not* required just to make more work for you, but to help you gain the above-mentioned insight into problem-solving. So, lavish some care on them.

Problem 1. The camel of **Khowârizm** (warmup problem, due October 5)

Let $\phi = (1 + \sqrt{5})/2 = 1.61803\ 39887\ 49894\ 84820\ \ldots$ be the "golden ratio."

A legendary camel in the ancient Persian valley of **Khowârizm** used to walk around and around on a circular road that was exactly one parathanha in circumference. Every time this camel would travel a distance of precisely ϕ **parathanhæ,** it would deposit some beautiful dung that shone like gold. The golden dung was, of course, bewitched; it **was** certain death to anyone who touched it or disturbed it in any way. Thus, **over** the years a series of such deposit6 continued to be built up along the road.

Everyone who passed that way noticed a strange thing-the golden cameldung seemed to be almost equally spaced. One could practically use the heaps as mileposts. In fact, at the time when there were n deposit6 on the road, the following condition was true (although only a few mathematicians knew it): There **was** a point x_n on the road such that if you started at x_n you would pass exactly one golden dung heap during the time it took to travel one nth of a parathanha. Repeating this n times until returning to x_n, you would see only one heap per segment *of* your tour. The mathematician6 of that time called this the "n-**fold** cyclic dung distribution property."

The magic camel never left the road, and it continued to make its enchanted deposits for many **years.** Every time n would increase by one, the n-fold cyclic dung distribution property remained valid. But one year, just a6 a new deposit was about to be made, a mysterious cloud appeared and enveloped the poor beast; and when the cloud moved away, the camel **was** gone. The spell was broken, and the gold could now be **freely** taken.

One of the mathematicians had anticipated this, since he knew that the n-fold cyclic dung distribution property would fail at the time of the next deposit. This brilliant **sorceror** had made preparations **for** the fateful moment, and his agents immediately pounced on the gold when the enchantment was lifted. He became **the** richest man in the valley, so he gave up mathematics and was assassinated two months later.

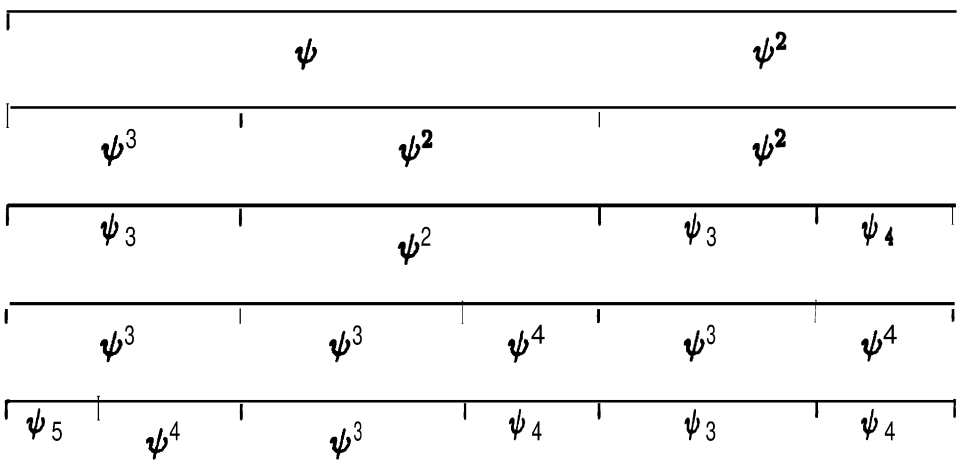
The problem is to determine x_n *for* $n = 1, 2, \ldots$; and to determine the final value of n when the camel vanished,

Notes from October 3

In the real world, the specifications for computer programs change and grow. To demonstrate this, DEK added a second camel to Problem 1; this camel appears after the first has disappeared and its n^{th} dung deposit is at position $(\lg(2n - 1)) \bmod 1$, where lg means logarithm to the base two. The sequence determined by these numbers has been studied by Lyle Ramshaw [On the *Gap Structure of Sequences of Points* on a *Circle,* STAN-CS-77-637, November 1977]. It is, in many respects, the most uniform distribution of points you can achieve if you do not know beforehand when you must stop placing points.

MH led off the discussion of Problem 1. He described his first stab at the problem as "a little loop to print elements of the sequence": to get a feeling for the general appearance of the dung distribution, he looked at the first few members of the sequence. Looking at the distribution, we digressed briefly to explore some of the properties of the sequence $\{n\phi \bmod 1\}$.

If we let $\psi = \phi - 1 = 1/\phi = \phi \bmod 1$ and draw the first few dung heap locations, labelling lengths between heaps, here's what we get:



Observe that each new dung heap falls into one of the longest intervals that existed before its deposition; further, new dung heap8 cut the interval into which they fall into piece6 whose lengths are in the golden ratio; finally, only three different length6 of inter-heap interval exist at any stage. For some elaboration of these facts, and further references, see the section on hashing (96.4) in Donald E. Knuth, *Sorting* and *Searching,* Volume *3* of *The Art of Computer Programming,* Addison-Wesley, 1974. (See also the frontispiece facing page 1 of that volume for an extended picture.)

Fascinating as these observations are, they don't seem to help us much with the problem of finding the value of n at which the n-fold cyclic dung distribution property fails. And, they are even less likely to help us solve the problem for sequences other than $\{n\phi \bmod 1\}$.

RTB initiated discussion of the next approach. He noted that our search for the x_n satisfying the n-fold cyclic dung distribution property need only consider possible values of x_n that lie in an interval shorter than $1/n$, since in some sense such intervals are the hardest to fit. DMO mentioned what he called a "brute force" method for finding an x_n if one exists: two nested loops, the outer loop cycling through possible values of x_n from one endpoint of a shortest interval to the other, the inner loop cycling around the circle checking whether each arc of length $1/n$ contains exactly one dung heap.

The problem with such an approach, as NCR pointed out, is that the outer loop can't cycle through all possible values for x_n: there is no way to choose a step size small enough to eliminate the possibility that you step over a possible value for x_n. Hence, this method might have the camel vanishing before its time.

MES proposed that instead of shifting a point around the circle in steps of length $1/n$ we shift an *interval* around the circle. Since the shortest inter-heap interval is always shorter than 1/n, we will start with such an interval, and after each shift, chop out parts of it that we know can't contain x_n because they don't see one heap each shift. Then, when the interval has come full circle, anything left is a possible value for x_n. This method has the advantage that it tells us the set of all possible x_n, rather than just one value.

At this point, DRM asked what kind of interval MES proposed shifting around the circle, and how this interval was to be specified. MES's answer that the interval was to be open, determined by its endpoints, prompted DCH to demand a clarification of the problem: does seeing a dung heap at the closed endpoint of an interval constitute passing it? In other words, if we started our x_n at a dung heap, and we passed or reached exactly one heap each $1/n$ parathanhae (the first having been *seen* at the very beginning of the journey), would that constitute an acceptable x_n? We took a vote and decided that it wouldn't, but then DCH pointed out that our decision didn't matter since if such a reference dung heap x_n existed, there would be an ϵ for which $x_n - \epsilon$ also had the n-fold cyclic dung distribution property, because a half-open interval is non-empty if and only if the corresponding open interval is non-empty. In any case, closed intervals won't work; *for* example, $x = 1/6$ is not legitimate when there are three heaps $\{0, 1/6, 5/6\}$.

DMO suggested a second method: imagine cutting the circle into n equal pieces starting at some arbitrary point. (This constitutes projecting each dung heap modulo 1/n.) DMO claims that if an x_n exists, one of the midpoints between two consecutive *projected* dung heaps must be a possible value, but he offered only a loose proof. DMO's process does suggest a nice picture for our task, though: imagine a device like an apple slicer with evenly spaced blades radiating from the center; placing the device on the circle and rotating it until each segment contain6 exactly one dung heap, if possible, is precisely our job.

DMO's idea also raises some interesting questions. Could one try just the midpoint6 between each pair of consecutive dung heap6 on the original circle, and if none satisfied the n-fold cyclic dung distribution property, be assured that no x_n exists? CPP claimed that the answer is no, and attempted to offer a counterexample, though he hadn't sufficient detail to present it on the spot. (Note: one counterexample is $\{0.0, 0.2, 0.3, 0.65\}$.) In h'is d'iscussion, CPP used the term "live zones" for regions of the circle that, so far as we know at a particular time, could contain a value for x_n.

A second interesting question is whether the actual live zone6 will always be connected subintervals of the circle.

DEK offered a couple of observations at this point. Although we'd spent a fair amount of time exploring conjectures not directly related to the problem, we shouldn't consider time so spent wasted, for two reasons: such digressions can offer us insight6 into the problem that prompted them, and they can be quite interesting in their own right. Research talent largely consists of the ability to ask interesting questions, not only to answer them. CPP's coining the term "live zone" for a concept that proves useful in talking about a problem is an excellent example of inventing terminology to help our minds cope with the complexity of problems; such vivid descriptive terms are often useful, and if our vocabulary is restricted to mathematical terms, our problem-solving ability is limited.

DEK wondered whether it was critical to start with the shortest interval in MES's method, or whether we could start with any interval. WOL (among others) claimed that it's not, because we'll eventually chop out any pseudo-live zones anyhow. A harder question is whether starting with the shortest interval buy6 us anything in the computation. In this connection, JJF pointed out that if we know the shortest interval at time n, it's easy to figure out which is shortest at time $n+1$, since each new point breaks up an existing interval; thus, we needn't search for the shortest interval each time n increases. So, an apparent source of inefficiency (searching for the shortest interval) may not be time-consuming if we have a suitable data structure.

DAS contributed another conjecture as the basis for his method of finding x_n. He claims that the intersection of the projections modulo $1/n$ of all inter-heap intervals that are shorter than l/n, if non-empty, is the live zone. However, others didn't believe this would be true.

Time was running out, so DEK recapitulated the questions that awaited answers: (1) Is the set of all possible x_n a connected interval modulo l/n? (2) Is DAS's conjecture true? He suggested that these questions be answered in the calm and privacy of our study rooms.

Notes from October 5

We began by considering DAS's conjecture that the intersection of the projections modulo l/n of all inter-heap intervals shorter than l/n is the live zone con-

taining possible x_n values. We had a fairly lively discussion going when DAS arrived and offered this proof:

Let the n pieces of dung be at road positions $d_1 < d_2 < \ldots < d_n$ and say $z_1 = x_n, z_2 = (x_n + \frac{1}{n}) \bmod 1, \ldots, z_i = (x_n + \frac{i-1}{n}) \bmod 1, \ldots, z_n = (x_n + \frac{n-1}{n}) \bmod 1$ are n points spaced equally around the circle. Now if $\delta_i = d_{(i \bmod n)+1} - d_i > 1/n$, there is always at least one point z_j between these dung heaps; so we have at least one point between every consecutive pair of heaps if we check this condition only when $\delta_i \leq 1/n$. Furthermore it suffices to check that there is at least one point between heaps, since the complementary condition ("at *most* one point between") follows from the fact that there are n points and n heaps.

Since the intersection of a finite set of intervals on the real line is again an interval, we might think at first that DAS's theorem gives us a very neat way to compute live zones. But JEB pointed out that since we're working with intervals on a circle, not a line, we might have to keep track of live zones composed of two disjoint intervals, when one interval "wraps around" to intersect both ends of another.

Next, DEK asked people to present solutions they were particularly happy with. CPP, CHT, and WOL were the first volunteers. If you recall that CPP coined the term "live zones," you should understand what he meant by saying that his team worked by determining the dead zones of the circle. Using the above notation, we can state their method this way: for each d_i, with $1 \leq i \leq n$, all points between $(d_{(i \bmod n)+1} - 1/n) \bmod 1$ and d_i are marked "dead"; also, their images under shifts by k/n are marked "dead." The union of the dead zones for $1 < i \leq n$ is the "master dead zone" whose complement contains all possible values for x_n.

Since this method requires that the immediate predecessor and successor of each dung heap around the circle be known, CPP, CHT, and WOL stored the heap locations in a linked list. A couple of observations made the implementation quite nice. We can do all the computation on the interval $[0, 1/n)$, since that will tell us about other dead zone6 around the circle implicitly: they're the image of the dead zone of $[0, 1/n)$ under shifting by some multiple of $1/n$. Also, it turns out that the dead zone only grows at either end, so it'6 easily represented by its endpoints.

The other team to present its method was WDG and DRR. Conceptually, their method is akin to the "apple slicer" method mentioned last time: they attempt to shift a comb with teeth 1/n apart around the circle until one heap lies between each consecutive pair of teeth. To avoid trying to rotate a circular comb they make two copies of the list of dung heap locations, hooking the second copy onto the end of the first. Then, they walk down the comb, noting conflicts (zero or more than one heap per comb interval) and, for each conflict, noting the minimum distance the comb must be shifted to repair the conflict. On reaching the end of the comb, they shift it by the maximum shift that is necessary. This walk-down-and-6hift process is repeated until the comb is positioned properly *or* the cumulative shift is greater

than l/n, the latter implying that there is no value for x_n.

DCH remarked that there's also a maximum possible shift to repair conflicts, and claimed that finding the minimum of the maximum possible shifts will give the set of all possible answers in one walk-down-and-shift iteration.

Anyhow, everyone agreed that the camel of **Khowârizm** disappeared when ready to make its fifteenth deposit, and the logarithmic camel vanished at the time of its twelfth. DEK mentioned that he had expected the camels to last a lot longer than they did and he posed the following question that is, as far as he knows, unsolved: What sequence maximizes the n at which the n-fold distribution property first fails? [The simpler question in which the distribution property is required to hold for $x_n = 0$ was solved by E. R. Berlekamp and R. L. Graham in *J. Number Theory* 2 **(1970),** 152-161: They showed that there is a sequence having the property with $x_n = 0$ for $1 \leq n \leq 17$, but no sequence does this for $1 \leq n \leq 18$.]

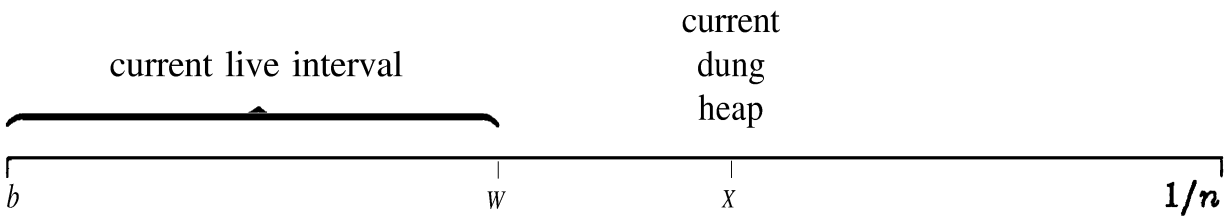
DRM asked about the programs' sensitivity to round-off **error.** This question is important because ϕ is irrational, hence not representable exactly on a finite digital computer. CHC mentioned that he'd run his programs with both single- and **double-**precision floating-point variables, and had noted no difference in the answers. While this is reassuring, and often the only thing we can do to alleviate round-off worries (at least with the available hardware), there is in this instance a way to avoid attempts at floating-point representation of ϕ altogether. Since we need only answer the question "Is $a + b\phi > 0$?" when a and b are integers, we can use this formula (derived by DRM):

$$\operatorname{sign}(a + b\phi) = \begin{cases} \operatorname{sign}(2a+b), & \text{if } a^2 + ab - b^2 > 0; \\ \operatorname{sign}(b), & \text{if } a^2 + ab - b^2 < 0. \end{cases}$$

DEK asked JEB to talk a little about his attempt to find an analytic way to determine the value of n at which the n-fold cyclic dung distribution property first fails. Working **from** methods *for* deriving the best rational approximation to ϕ whose denominator is bounded above by **some** number, he hoped to compose a predicate of the form "$|\phi - j/k| < \epsilon$ implies that the property fails for **some** n between k and $2k$." Although he didn't succeed, and such a method would not help with sequences that are not multiples of irrational numbers (like the logarithmic sequence), this effort qualifies as a good example of not turning to the wmputer before thinking enough about a problem ourselves. DEK remarked that the quality of research at a university sometimes seems inversely proportional to the quality of available computing resources.

Next, DEK presented his thoughts about a good solution; his solution was essentially like that of CPP, CHT, and WOL. He remarked that it seems best to keep the d_i locations in a simple array. A clever data structure like a balanced binary

tree shouldn't be used unless it were clearly superior to a simple table, since the overhead involved is quite substantial until n gets into the hundreds. He decided to work in the following framework: (The left endpoint *of* the current live zone is b.)



Here is **DEK's** two-part loop invariant: up to position k, (1) all solutions x_n are in the interval $(b, b + w)$ modulo $1/n$; and (2) $d_k \equiv (b + x)$ (modulo $1/n$). At $k = 0$, this invariant holds for $b = 0$, $w = 1/n$, and $x = 1/n$. The following steps will maintain this invariant, assuming that $0 = d_0 < d_1 < \cdots < d_n = 1$:

```
k ← 0;
while k < n do
    begin
    k ← k + 1;
    δ_k ← d_k − d_(k−1);
    if δ_k ≤ 1/n − x
        then quit   [live interval too short for dung heaps so close together]
    else if δ_k ≤ w + 1/n − x
        then x ← w ← δ_k − (1/n − x)   [chop off right end of live interval]
    else if δ_k ≤ 2/n − x
        then x ← δ_k − (1/n − x)   [move heap closer to origin]
    else if δ_k ≤ 2/n
        then begin   [chop off left end of live interval]
             w ← w − (δ_k − 2/n + x);
             b ← b + (δ_k − 2/n + x);
             x ← 1/n
             end
    else quit   [heaps are further than 2/n apart]
    end
```

This can be made much more elegant by trying to combine cases so as to reduce the nesting of conditional statements. For example, if we include "$w > 0$" in the while-condition, we can ignore the first and last cases, since the arithmetic we'll do will make w negative and stop the loop next time around.

Notes from October 10

DEK started off by explaining the significance of the name **"Khowârizm": Abu** Ja'far Mohammed ibn **Mûsâ al-Khowârizmî** was the Persian author of ***Kitab*** al ***jabr w'al-muqabala,*** and our word "algorithm" was derived from his last name.

Next, DEK showed us **JEB's** one-line solution to Problem 1. We have a set of points

$$0 = d_0 < d_1 < \cdots < d_{n-1} < d_n = 1,$$

and we want to know a value x_n such that

$$d_0 < x_n < d_1 < x_n + \frac{1}{n} < d_2 < \cdots < d_{n-1} < x_n + \frac{n-1}{n} < d_n.$$

We can write this more compactly as

$$d_k < x_n + \frac{k}{n} < d_{k+1}, \qquad 0 \le k < n, \tag{*}$$

and this set of inequalities can be stated succinctly as

$$\max_{0 \le k < n} \left(d_k - \frac{k}{n} \right) < x_n < \min_{0 \le k < n} \left(d_{k+1} - \frac{k}{n} \right).$$

This solution is evidently quite a bit simpler than the method DEK showed us last time. We can draw several lessons from this experience. One is that looking at the detailed (algorithmic) structure of a problem may not lead to the most elegant solution. We should stay on a higher level of abstraction when we can. A more general observation is that we should try to have as many ways to look at a problem as possible, since the larger perspective they give us can improve our solution considerably.

DRF mentioned that for each k, only one of the inequalities in (L) gives us any interesting information; that is, for each k, either $d_k - \frac{k}{n} < 0$ or $d_{k+1} - \frac{k}{n} \ge \frac{1}{n}$. Thus if $d_k - k/n \ge 0$ we needn't bother to update the current **"min"** while computing **(*).** WOL pointed out that the same number of comparisons is being made in this method as in the interval-shrinking methods. This is true, but the algebraic formulation increases the method's perspicuity.

We finished off camel problems for this quarter with DEK presenting a method for solving the inequalities **(*)** assuming one is given the locations h_i, $1 \le i \le n$, of n dungheaps. The interesting feature of this method is that it **doesn't require** sorting; in fact, given any $m > n$, it will work in $O(m)$ steps. For $0 \le i < m$, we

keep the following numbers:

$$a_i = \min\{h_k \mid \lfloor mh_k \rfloor = i\}$$
$$b_i = \max\{h_k \mid \lfloor mh_k \rfloor = i\}$$
$$c_i = \|\{h_k \mid \lfloor mh_k \rfloor = i\}\|$$
$$d_i = c_0 + c_1 + \cdots + c_{i-1}.$$

Then, using the d_k's to represent the sorted h_i as before, we have

$$\max_{0 \le k < n} \left(d_k - \frac{k}{n} \right) = \max\left(\max_{c_i \neq 0} \left(a_i - \frac{d_i + 1}{n} \right), 0 \right)$$

$$\min_{0 \le k < n} \left(d_{k+1} - \frac{k}{n} \right) = \min\left(\min_{c_i \neq 0} \left(b_i - \frac{d_{i+1} - 1}{n} \right), \frac{1}{n} \right)$$

[Reference: Teofilo Gonzalez, Sartaj Sahni and W. R. Franta, "An Efficient Algorithm for the Kolmogorov-Smirnov and Lilliefors Tests," *ACM* Trans. *on Mathematical* Software 3 (1977), 60–64.]

Notes **on Solutions to Problem 1**

People used three kinds of algorithm to solve this problem. Those who viewed the problem as one of solving a system of $2n$ simultaneous inequalities generally coded their programs more clearly than those who were shifting or trimming some interval, even though both methods use the same algebraic operations.

Most people received an "A" on algorithm. Those who didn't had gone to extra trouble that was unjustified by the problem (e.g. using a fancy data structure), or used a method that was inferior to either of the methods described above. No one did worse than a "B" on algorithm.

Code style was generally good. (One person did turn in a solution with no writeup and very few comments, though.) Most of you don't need to be harangued about documenting your programs, but it was pretty obvious who had structured comments into their programs and who had tacked them on after getting the program working. A careful approach to commenting is important; don't write them just to please the grader.

Most people made good use of procedures, passing parameters around to keep pieces of the program independent. But when a procedure affects global variables, it should be clear what it's doing to them. Such steps need to be carefully commented so they don't appear mysterious to someone reading your program.

No one received above a "B" on code style. This is partly because no one writes code in exactly the same style as the grader does, but more importantly because there's plenty of room for everyone to improve his or her style. The grader tried to

indicate things he found about your programs that could be improved. If you had too *few* helpful comments or didn't use procedures you didn't get a **"B,"** either,

The biggest concern is the quality of the writeups, Most people just handed in a description of the solution method they'd chosen and how their programs worked. The writeups are supposed to be more: they should describe your *progress* **toward** *solution.* This includes false starts you made and blind alleys you followed. (Explain these, don't just mention them. At least two people said something like **"We** spent a lot of time studying .... However, this was useless," and dropped the subject.) You should also explain why you tried a particular approach, even if your reason **was** only a feeling or hope that it would work. Such introspection will help you to learn research techniques in general (the main goal of this course).

Look at it this way: you can explain your answer to a problem to someone else, and then he or she may be able to explain it to anyone. But only you can explain **how you arrived at the** solution you **did,** So spend some time looking over your work on the problem, then write a description of the main stages of thought you went through on your way to the solution you hand in. Don't be afraid: there's no such thing as a stupid approach to a problem. The difficulty is probably that you equate being wrong with being stupid.

The writeups should also include any discoveries you made while working on the problem. For example, one person remarked that round-off **error** caused him trouble in representing the master dead interval. **Another** thing these writeups should include **is** questions that remain now that your work on the problem has, for the time being, ended.

Here is a summary of the solutions people handed in. The short description of methods tells the closest relative that a team's approach had of methods shown in class. The column of distinctive features tells one of the things that distinguished a solution from the others; as you can see, some are good and others are not.

TEAM	**METHOD**	**DISTINCTIVE FEATURES**
RTB,AFB	interval shifting	presented two approaches
JEB,MH	inequality solution	presentation polished to a high gloss
DCH,ARR	interval trimming	humorous presentation
JJF	interval trimming	attempted **proof of correctness**
DRF	interval trimming	very short program
WDG,DRR	rotating comb	excellent writeup
MSK	interval trimming	external dung calculation procedure
MAI	inequality solution	mentioned ideas in writeup
NCR	?	paucity of documentation
DMO,AMY	interval trimming	excellent writeup

CPP,CHT,WOL	interval complementation	very careful presentation
MES	interval trimming	small bug
DAS	interval complementation	nice writeup
IAZ	interval shifting	nice code

Problem 2. Bit fiddling and image processing (due October 19)

In this problem we will be dealing with pictures represented on a discrete raster. We have an $n \times n$ grid consisting of square "pixels," each of which is white or black. Using 0 to represent white and 1 to represent black, we will be able to represent the picture compactly with 36 pixels per word (on the AI Lab computer), and the computer's Boolean operations will facilitate the manipulation of pictures. There are two parts to this problem: first to "clean up" a picture, and second to do edge following that describes the boundaries of the "objects" in a cleaned-up picture.

A. The cleaning-up phase is intended to change stray black pixels to white and vice versa, as well as to remove sharp turns of more than 90° at the edges of black areas.

For purposes of discussion, a picture P will be regarded as the set of all black pixels in some image, and the complementary set P^{-} will be the set of all white pixels. We define the closure P^{C} of P to be the smallest picture containing P such that every white pixel (i.e., *every* element of P^{C-}) is part *of* a 3×3 square of white pixels. A picture is closed if it is equal to its closure, The *interior* P^{O} of P is defined to be the largest picture contained in P such that every black pixel is part of a 3×3 square *of* black pixels. A picture is *open* if it is equal to its interior.

Examples: (Imagine infinitely many white pixels surrounding these diagrams)

$$P = \begin{matrix} 1\,1\,0\,0\,1\,1 \\ 1\,1\,0\,0\,1\,0 \\ 0\,1\,1\,1\,1\,0 \\ 0\,1\,1\,1\,0\,0 \\ 0\,0\,1\,0\,0\,0 \end{matrix} \qquad P^{C} = \begin{matrix} 1\,1\,1\,1\,1\,1 \\ 1\,1\,1\,1\,1\,0 \\ 0\,1\,1\,1\,1\,0 \\ 0\,1\,1\,1\,0\,0 \\ 0\,0\,1\,0\,0\,0 \end{matrix} \qquad P^{CO} = \begin{matrix} 0\,1\,1\,1\,1\,0 \\ 0\,1\,1\,1\,1\,0 \\ 0\,1\,1\,1\,1\,0 \\ 0\,1\,1\,1\,0\,0 \\ 0\,0\,0\,0\,0\,0 \end{matrix}$$

The first subproblem is to replace a given picture P by P^{CO}, the interior of its closure. We will say that a picture is clean if it equals P^{CO} for some P. Write a program that finds P^{CO}, given a 72×72 picture P.

While solving this problem you may wish to develop a theory along the following lines. Let us write

$x \prec y$ if $\text{row}(x) \leq \text{row}(y) \leq \text{row}(x) + 2$ and $\text{col}(x) \leq \text{col}(y) \leq \text{col}(x) + 2$;
$x \succ y$ if $\text{row}(x) - 2 \leq \text{row}(y) \leq \text{row}(x)$ and $\text{col}(x) - 2 \leq \text{col}(y) \leq \text{col}(x)$.

Furthermore we define

$$P^{\prec} = \{x \mid x \prec y \text{ for some } y \text{ in } P\};$$
$$P^{\succ} = \{x \mid x \succ y \text{ for some } y \text{ in } P\}.$$

In these terms, prove *or* disprove the following statements:

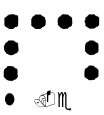
(1) $P^{-C-} = P^{O}$.
(2) P^{O} is the union of all 3 × 3 square black pictures contained in P.
(3) x $\prec y$ if and only if $y \succ x$.
(4) $P^{\prec}$ is open.
(5) $P^{O} = P^{-\prec-\succ}$.
(6) If P is a picture such that every black pixel is included in a set of 3 consecutive black pixels in the same row and in a set of 3 consecutive black pixels in the same column, then P is open.
(7) If P is a clean picture and if x, y, z are three consecutive elements of a row, where x $\in P, y \notin P$, and $z \in P$, then the pixels immediately above and below y are not in P.
(8) If P is a clean picture and if w, x, y, z are *four* consecutive elements of a row, where w $\in$ P, x $\notin P$, $y \notin P$, and $z \in P$, then the pixels immediately above and below x and y are not in P.
(9) A clean picture is both open and closed.

B. The ***edge-tracing* phase** is going to provide *a* description of a clean picture that grows linearly (instead of quadratically) with the resolution of the raster.

Let P be a clean picture. We **call** the black pixel x a boundary ***point*** of P if at least one of x's four neighbors *(above,* below, left, or right) is white. The problem in this phase is to group the boundary points into "closed cycles *of* king moves." In other words, if there **are m boundary points, we want to arrange** them **into** p sequences

$$x_{11}, x_{12}, \cdots, x_{1m_1}$$
$$\cdot$$
$$x_{p1}, x_{p2}, \cdots, x_{pm_p}$$

such that $m_1 + \cdots + m_p = m$ **and** each $x_{i(j+1)}$ **is a king** move **away from** x_{ij}; **also** x_{i1} **is a king** move *away* **from** x_{im_i}. A king move **is a** move one step up, down, left, right, or diagonally. **In** the **example** *above,* **all** eleven of the boundary points belong to a single cycle



of king moves, but in general we will have several cycles if there are several disconnected areas *of* black, or if there are "holes."

The cycles of king moves can be still further refined, since it turns out that a clean picture has cycles with the following property: After a diagonal move, the path either stays in the same direction or branches 45° to the left or right of its current direction. After a rectilinear move, the path either stays moving in the same direction or branches 45° or 90° to the left or right of its current direction. Thus we can encode each cycle as a compact sequence of instruction pairs "(number, turn)", meaning to continue in the same direction for the given number of steps and then to turn in the specified way. When moving diagonally, the next turn will either be *L* or *R* (45° left or right); otherwise it will be either *LL* or *LR* or *RL* or RR (90° left, 45° left, 45° right, 90° right). We may assume that the path begins in the upward direction in the leftmost column of the cycle; thus, the example boundary above can be encoded by the sequence *of* king-move instructions

$$(3,RR),\quad (3,RR),\quad (2,RL),\quad (1,R),\quad (2,RR).$$

(The final turn "RR" is redundant and could be omitted.) Together with the coordinates *of* the starting point, this sequence of instructions completely defines the cycle.

You are to write a program that produces such instruction sequences for the boundary cycles of the clean 72 × 72 pictures produced by your first program. You should also prove (informally) that your program will handle all clean pictures.

Input to your programs will be from a file of 72 lines each of 72 characters ("0" or "1"). To compute with these in your programs you should try to use 144 36-bit words (integers).

Hans Moravec's graphic display routines will be used to display your programs' output. Screen dimensions 504 × 504 allow you to display each black (resp. white) pixel as *a* black (resp. white) 7 × 7 rectangle. You should show the border-tracing king moves by drawing lines that darken light areas and lighten dark areas of the picture between the centers of rectangles.

Your solutions will be run on a data file that you cannot see beforehand, although a test data set is supplied. The grader will expect to be able to look at (1) the original picture, (2) its closure, (3) the interior of its closure, and (4) the king moves.

Notes from October 5

DEK introduced the second problem as a good opportunity to develop a theory before sitting down to write a program. He also pointed out that at least one of the statements to prove or disprove is false. So, we see that the process of developing a theory can profit from **a** healthy skepticism about any statement alleged to be true: in fact, it's often better to try first to **disprove** a statement by looking for counterexamples. This searching may help us understand the question better, hence give us ideas about how to prove the statement if it is true.

Notes **from October 10**

DEK remarked that he had tried to state his definitions in analogy with topological notions of openness and **closedness.** This analogy was designed to help, not confuse, people thinking about the problem.

We started off trying to prove conjecture (1) $P^{-C-} = P^{O}$. People took essentially two approaches: some tried to reason about the white and black pixels, while others approached the problem in a purely algebraic framework, In this instance, it proved more fruitful to look only at the symbols and not be concerned with their meanings in terms of bits. For example, NCR observed that the definitions of "interior" and "exterior" are complementary: the closure P^{C} of P is the smallest $Q \supseteq P$ such that every element of Q^{-} is part of a 3×3 square in P^{-}. If we change "smallest" to "largest" and invert the subset and complementation operators, we arrive at the definition of the interior of P as the largest picture $Q \subseteq P$ such that every element of Q is part of a 3×3 square in P.

This observation offers a way to follow SOY's suggestion that we try to prove $P^{O-} = P^{-C}$. (If we can do that, complementing both sides gives the result we seek,) **We** know P^{O-} is the complement of the largest $Q \subseteq P$ such that every element of Q **is** part of a 3×3 square in P; let's write P^{O-} as Q^{-}. Inverting as in the definitions, we can say P^{O-} is the smallest $Q^{-} \supseteq P^{-}$ such that every element of Q is part of a 3×3 square in P. But this *says* precisely that Q^{-} is the closure of P-, so $P^{O-} = P^{-C}$.

Applying the definitions very carefully and using standard methods from logic, we have deduced a relationship that might not have been apparent if not expressed this way. An imagination for pictures is often helpful, but formulas can reveal patterns to us that we wouldn't see otherwise, Part of the way they do this is by suppressing details about the particular problem (e.g., just what is meant by "smallest" and "largest"). Our experience with camel dung revealed one such blind spot.

DRF pointed out that the above relationship is false if the plane of pixels is bounded (Le., if P^{-} really means $F \cap P^{-}$ *where* F is some finite frame). DEK

apologized *for* implying that all pixels outside the **frame** of reference are white; they all might also be black, The important point was that our formula $P^{O} = P^{-C-}$ has been proved on the infinite plane, and we are free to use it to derive other formulas valid in finite frames. This led to the question: under what circumstances can we conclude that $P \subseteq F$ implies $P^{C} \subseteq F$? Before answering this question, we came face to face with the fact that P^{C} might not be well defined; what is the "smaller" of two incomparable sets?

As class drew to a close, we polished off **a** proof of property (2): P^{O} is the union U of all 3×3 square black pictures contained in P. (This property assures us that interiors are well defined, hence by (1) the closure is also well defined.) AFB observed that $P^{O} \supseteq U$ is easy: any pixel x that is part of a 3×3 black square contained in P is in P^{O} by definition. DMO offered this proof of the other inclusion: if $x \in P^{O}$, then x is **in a** 3×3 black square contained in P (otherwise we could delete x and get a smaller set satisfying the criterion), so $x \in U$.

Returning to our frame question, when can we conclude that $P \subseteq F$ implies that $P^{C} \subseteq F$? **MSK** observed that this is true if F is closed. **WOL noted** that rectangles are closed. (After class, DEK noted further that the implication fails **for-P** $= F$ if F is *not* closed; hence the implication is valid for all P if and only if F is closed.)

Finally, let's observe that we can prove the above implication as a special case of the **more** general rule $P \subseteq Q$ implies $P^{C} \subseteq Q^{C}$. Property (2) makes it clear that $P \subseteq Q$ implies $P^{O} \subseteq Q^{O}$ since the squares in the union for P^{O} are in the union for Q^{O}. Now $P \subseteq Q$ implies $P^{-} \supseteq Q^{-}$ implies $P^{-O} \supseteq Q^{-O}$ implies $P^{-O-} \subseteq Q^{-O-}$, i.e., $P^{C} \subseteq Q^{C}$.

DEK posed a final question: Is the intersection of closed pictures closed?

Notes from October 12

This problem is motivated by the need to compute for high resolution graphic display devices: How can we represent a picture so that doubling the resolution doesn't quadruple the number of bits we need? At first, this question might not seem **to** make much sense: If any pixel in an $m \times n$ raster can be "0" or "1" independent of the others, won't we need mn bit6 t o distinguish the 2^{mn} possible patterns?

One answer is that the pictures we represent will be **large compared to the size** *of* one pixel. This leads us to hope that the pictures are relatively smooth, so that a pattern like

```
1 0 1 0 1 0
0 1 0 1 0 1
1 0 1 0 1 0
0 1 0 1 0 1
1 0 1 0 1 0
0 1 0 1 0 1
```

is far less likely to occur in the pictures than large areas of ones or zeros. Our definitions of closure and interior are intended to capture this notion of smoothness. Using them, we can clean up our pictures by removing stray white or black pixels; then, since we can represent clean pictures fairly compactly by specifying only the boundaries of the regions, we hope that the number of bits we need will increase only linearly with the resolution of the raster, rather than quadratically.

DMO asked if we'd ever need to make **90°** left turns if we followed the directions in the problem statement and traced around the boundary in the clockwise direction. This question is prompted by the fact that the starred bit in the example below is not on the boundary: we'll take two **45°** left turns negotiating that corner.

```
0 0 0 1 1 1
0 0 0 1 1 1
0 0 0 1 1 1
1 1 1 * 1 1
1 1 1 1 1 1
1 1 1 1 1 1
```

CHT pointed out that we'll have to make **90°** left turns if we have "holes" in our pictures. For example, there'll be one at the starred position below. On the other hand, we'll never need to make **90°** *right* turns when tracing the boundary of a hole inside a region.

```
0 0 0 0 0 0 1 1 1 1 1 1
0 0 0 0 0 0 1 1 1 1 1 1
0 0 0 0 0 0 1 1 1 1 1 1
0 0 0 1 1 1 1 1 1 1 1 1
0 0 0 1 1 1 1 0 0 0 1 1
0 0 0 1 1 1 1 0 0 0 1 1
1 1 1 1 1 1 * 0 0 0 1 1
1 1 1 1 0 0 0 0 0 0 1 1
1 1 1 1 0 0 0 0 0 0 1 1
1 1 1 1 0 0 0 0 0 0 1 1
1 1 1 1 1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1 1 1 1 1
```

We discussed briefly the idea of changing the definition of boundary point, but two facts dissuaded us *from* doing so. One was **DEK's** admonition that we should try to get away with as few boundary points as we can. The other was **MSK's** example, where the starred point, if considered to be on the boundary, makes trouble for our cycles of king moves: they could intersect themselves, violating the specifications of the problem.

1 1 1 0 0
1 1 1 0 0
1 1 * 1 1
0 0 1 1 1
0 0 1 1 1

DEK asked at this point whether anyone knew of *an* example of a picture with two different legal **sequences** of king moves that describe it. No one answered, but notice that the last example is such a picture! [One of the sequences was made illegal on October **17.**]

The first fact we proved today was (3) x $\prec y$ if and only if $y \succ$ x. NCR called his pictorial argument for this a proof "by exhaustion," Since x $\prec y$ means that x lies in the upper left of a 3 × 3 square containing y, while $y \succ$ x means y lies in the lower right of a 3 × 3 square containing x we have:

$x \prec y$: $\begin{matrix} x & y & y \\ y & y & y \\ y & y & y \end{matrix}$ $\qquad$ $y \succ x$: $\begin{matrix} x & x & x \\ x & x & x \\ x & x & y \end{matrix}$

If you look at the pictures, it does seem to be belaboring the obvious to try to prove **this.** Nevertheless, DEK did point out that if (3) *weren't* true, we'd have chosen a horrible notation, bound to cause us trouble.

MES proved (4) $P^{\prec}$ is open by rewriting $P^{\prec}$. *If we* define L(y) = {x | x $\prec y$}, we have $P^{\prec} = \bigcup_{y \in P} L(y)$. Notice that each L(y) is an open 3 × 3 square, so $P^{\prec}$ is the union of such squares, hence is itself open (by (2)). Incidentally, using complementation we can show that the intersection of closed pictures is closed.

MES pointed out that $P^{\prec}$ **consititutes** a smearing of P toward the top and the left. This means that we can compute $P^{\prec}$ in place if we proceed through the picture from top to bottom and left to right. DCH observed that the smearing may cause $P^{\prec}$ to have fewer components than P: separate regions in P may be smeared together in $P^{\prec}$.

We forged ahead to a proof of (5) $P^O = P^{-\prec\succ}$. Several people contributed to this, including WDG and JEB. Once again, we let the formal logical operations do a large amount of the work for us.

$$\begin{aligned}
P^{-\prec} &= \{x \mid \exists y \in P^-: x \prec y\} \\
&= \bigcup_{y \in P^-} L(y) \\
P^{-\prec-} &= \{x \mid \forall y \in P^-: x \not\prec y\} \\
&= \{x \mid \forall y: y \notin P \Rightarrow x \not\prec y\} \\
&= \{x \mid \forall y: x \prec y \Rightarrow y \in P\} \\
&= \{x \mid \forall y: y \in R(x) \Rightarrow y \in P\} \\
&= \{x \mid R(x) \subseteq P\},
\end{aligned}$$

where R(x) $= \{y \mid x \prec y\}$. Thus

$$\begin{aligned}
P^{-\prec\succ} &= \{z \mid \exists x \in P^{-\prec-}: z \succ x\} \\
&= \{z \mid \exists x: R(x) \subseteq P \wedge z \succ x\} \\
&= \{z \mid \exists x: R(x) \subseteq P \wedge z \in R(x)\} \\
&= \bigcup_{x: R(x) \subseteq P} R(x) \\
&= P^O.
\end{aligned}$$

DEK made two observations. We should all be comfortable with the logical notation and operations used above, As we've said many times before, it's important to be able to look at a problem from many different perspectives. Another thing to notice is that we haven't used the fact that the definitions are stated in terms of 3 × 3 squares of pixels as the basic clean units. So, our proof is valid for any definition that satisfies the more abstract notions embodied in (1) to (5).

DEK mentioned that he could have chosen a more symmetric relation like

$$x \sim y: \quad \begin{matrix} y\ y\ y \\ y\ x\ y \\ Y\ Y\ Y \end{matrix}$$

whose corresponding operation on a picture P smears points in P out in all directions. Using similar operations to the ones above, we can show $P^O = P^{-\sim-\sim}$, then use fact (1) to obtain $P^C = P^{-\sim-\sim-\sim}$. One problem with this definition is that it can't be extended to use squares of even dimensions as the basic clean elements. Another is that it's probably not as amenable to computation in place as the asymmetric relations $\prec$ and $\succ$.

As class ended, MES asked if we could reduce the size of the pictures to 70 × 70 since the $\prec$ operation may smear pixels out of the 72 × 72 frame given. DEK suggested that we look instead for a way to compute $P^{\prec}$ without using the extra border rows; is it possible?

Notes from October 17

JJF showed us this counterexample to (6) if P is a picture such that every black pixel is included in a set of 3 consecutive black pixels in the same row and in a set of 3 consecutive black pixels in the same column, then P is open. The starred (black) cell meets the row and column criterion, but this picture is not open, since that cell is not in a 3 × 3 square *of* black cells.

```
1 1 1 0 0 0
1 1 1 * 0 0
1 1 1 1 1 1
0 0 0 1 1 1
0 0 0 1 1 1
```

This was a sobering reminder that conjectures we make are usually false; textbooks give a misleading impression. We should always make conjectures, but give them a jaundiced eye, Our work trying to disprove conjectures is usually not wasted, since it gives our mind useful exercise in the problem area.

What about (7)? "If P is a clean picture and if x, y, z are three consecutive elements of a row, where $x \in P$, $y \notin P$, and $z \in P$, then the pixels immediately above and below y are not in P." We took a vote: four believed it, five were opposed, and twelve abstained. ARR asked whether the hypotheses could be satisfied, since if they couldn't, (7) would be vacuously true, JEB offered this example of a clean picture with three elements in the black-white-black pattern:

```
1 1 1 0 0 0 0
1 1 1 0 0 0 0
1 1 1 0 1 1 1
0 0 0 0 1 1 1
0 0 0 0 1 1 1
```

Here is JEB's proof of (7). Assume we have a configuration like this where the color of x's is not yet specified:

x x x
1 0 1.
x 1 x

We know that for the picture to be open, each 1 must be part of a line of three consecutive l's, so we have

x x x x x x x
1 1 1 0 1 1 1
x x x 1 x x x.
x x x 1 x x x
x x x 1 x x x

We can argue that one *of* the columns on either side of the central column must be black, again since this picture is open; without loss of generality, we assume the right side:

x x x x x x x
1 1 1 0 1 1 1
x x x 1 1 x x :.
x x x 1 1 x x
x x x 1 1 x x

Finally, we must add another column of l's on either side of the central column. In either location, thie column makes the 0 an element of *a* 3 × 3 square *of* black pixels, so it would have been blackened by the closure operation used to clean the picture, and not whitened again by taking the interior.

Our proof of (8) was similar. Assuming that P contains the configuration

x x x x
1 0 0 1,
x x 1 x

we can offer a very similar sequence of steps to show that the 0 above the 1 must actually be a 1.

DRM asked if we knew that some finite sequence of closure and interior operations would terminate, i.e. after taking closures and interiors for a while, would we

sfop changing fhe picture? JEB proved fhat the sequence always terminates very quickly: $(Q^{CO})^{CO} = Q^{CO}$. (Laundering a clean picture doesn't make it any cleaner.)

JEB's proof for this again relied exclusively on algebraic manipulations. We start with the fact fhat $Q^{CO} \subseteq Q^{C}$. Taking the closure of both side6 we have $Q^{COC} \subseteq Q^{CC}$; we certainly hope that $Q^{CC} = Q^{C}$, and if we assume it we can fake inferiors on both side6 to obtain $Q^{COCO} \subseteq Q^{CO}$.

IAZ proved fhe inclusion in fhe opposite direction: We know that $Q^{CO} \subset Q^{COC}$ since closures *grow*. Hence $Q^{COO} \subset Q^{COCO}$. This fime we assume fhaf fhe desirable relation $Q^{OO} = Q^{O}$ is frue, and we have our result.

Now, all we need is to prove the lemma that $Q^{OO} = Q^{O}$ for any Q. We caught ourselves already assuming this in several arguments, without proof, (By complementafion, $Q^{OO} = Q^{O}$ will give us $Q^{CC} = Q^{C}$ as well.) Buf now we can appeal to (2) to write $Q^{OO} = \bigcup_{x:R(x)\subseteq Q^{O}} R(x)$. If $R(x) \subset Q$ then $R(x) \subseteq Q^{O}$ by definition, and $R(x) \subseteq Q^{O}$ clearly implies $R(x) \subset Q$, so we can rewrite fhe condition on the union as $Q^{OO} = \bigcup_{x:R(x)\subset Q} R(x) = Q^{O}$.

DEK observed that we now have proved thaf a given picture lead6 to at mosf 14 pictures under the operations of —, O, and C: P, P^{O}, P^{O-}, P^{O-O}, $P^{O-O-} = P^{OC}$, P^{OCO}, P^{OCO-}, P^{-}, P^{-O}, $P^{-O-} = P^{C}$, P^{CO}, P^{CO-}, P^{CO-O}, P^{COC}. Applying —, 0, or C to any of these gives another in the set, Thus the "Kuratowski complement problem" in topology ha6 its analog in our system. [C. Kurafowski, "Sur l'opération Ā de l'analysis situs," Fundamenta Math 3 (1922), 182–199.]

DRM suggested that we might like to define a point in $P^{C} \setminus P^{O}$ to be grey, since we seemed to need to refer to such points fairly often in these arguments.

Fact (9), "a clean picture is both open and closed" turned ouf to be false too. Today'6 second example is a clean picture fhat is nof closed: the point between the block6 flashes on and off repeatedly when we fake closure and inferior. DEK said he had tried unsuccessfully to find an algorithm fhaf produces an open-and-closed picture "nearest" to a given picture.

DCH suggested fhat a different notation for the boundary-tracing king move6 might be useful, especially because fhe one suggested in fhe notes uses direction relative to one'6 current position and direction of movement, so that errors send fhe king moving to wildly different place6 fhan if should go. He suggested defining fhe king move6 with respect to a fixed coordinate system like compass directions. This scheme requires one more bif fhan the problem sfatemenf's method, buf it ha6 the advantage that one may choose to begin fhe edge tracing at any point. DRF suggested using the actual coordinates of corners to denote the edges in the boundary.

DCH explained hi6 algorithm for boundary tracing a6 "walking around so a6 to keep white to hi6 left at all times." He did not have fime to offer a proof that this works for clean pictures, Another algorithm needs to be used to gef from one closed

cycle to another in the picture.

DRM mentioned that the boundary edges can be found by a simpler scheme if instead of doing king moves we consider the boundary lines between pixels. This neat idea is described in the following report that can be found in the CS library: B.W. Lindsay and D.R. Morrison, "Algorithms to link scan data into ordered contours and topological algorithms." Sandia Corporation, **SIDCAP** report no. 5, **SC-DR-68-815** (January 1973).

Notes from October 19

We spent this class session discussing the solutions you were about to hand in. **DRF** was the first to volunteer to present his method. He had used some of the results we've proved (like $P^{O} = P^{-\prec\succ}$), but had devised an operation that made calculating $P^{\prec}$ easier, Let $P^{\rightarrow}$ be the picture that results from smearing each black pixel right two pixels, and $P^{\downarrow}$ the picture resulting from smearing black pixels down two; then we have $P^{\prec} = P^{\rightarrow\downarrow}$. This "factored" formulation is simpler because we can smear pixels along the same row or column without worrying about adjacent rows or columns: we've reduced a fundamentally two-dimensional problem to a **one-**dimensional one. Furthermore, less total work is involved (four OR's instead of eight).

DRF showed us a way to calculate $P^{\rightarrow}$ by walking back along a row. He showed that we wouldn't even need to look at each pixel in the row twice, since we don't need to check whether to smear if we've just smeared a black pixel. RTB pointed out that this process is equivalent to **logical-ORing** a row or column with itself shifted left one pixel and two pixels. His observation points up two different ways of evaluating bit-fiddling algorithms: counting serial bit operations or counting parallel ones.

Extending the notation some, so that we smear either l's or O's, **DRF** wrote $P^{C} = P^{\rightarrow_1\downarrow_1\leftarrow_0\uparrow_0}$. We can commute the operations on O's, giving $P^{C} = P^{\rightarrow_1\downarrow_1\uparrow_0\leftarrow_0}$. The middle two operations in this expression $(\downarrow_1\uparrow_0)$ constitute sort of a one-dimensional closure: one or two consecutive O's become l's; three or more remain 0.

DRF claimed that he didn't need to keep extra rows along the edges, but MES and WOL gave an example to show that he does, A 3 × 3 black square located two pixels from the edge of the picture smeared right will smear up against the border. But the only thing to distinguish those right-smeared bits from a 3 × 3 square that abuts the border in the original picture is the bits in the border: we need them when smearing white pixels left.

Anyhow, we can compute $P^{CO} = P^{\rightarrow_1\downarrow_1\uparrow_0\leftarrow_0\rightarrow_0\downarrow_0\uparrow_1\leftarrow_1}$ in five operations: the second and third and sixth and seventh are respectively the one-dimensional closure mentioned above and the corresponding one-dimensional interior operation. Furthermore, the fourth and fifth represent the extension *of* O's by two pixels in both directions. This discussion raised a question about the commutativity of these operators, since we know that $\rightarrow_0\downarrow_0$ equals $\downarrow_0\rightarrow_0$. JEB observed that $\rightarrow_0\downarrow_1$ is not the same as

$\downarrow_1\rightarrow_0$: the first makes a diagonal line of three 1's all white, while the second leaves a 1.

MES contrasted P^{OC} with P^{CO}: since the interior operation leaves all pixels outside the frame white and the closure is known to stay inside a rectangular frame, P^{OC} can be computed without extra rows to remember black pixels that stray outside and then disappear; we know that P^{CO} may require extra rows, however. Was there any reason for picking one over the other as our definition of clean picture? There was no response, but DEK mentioned later that there is indeed a reason, since you can't always trace the boundaries of a P^{OC} by disjoint king moves. (The pattern 010 can occur.)

ARR observed that the simple pattern below behaves rather differently under the two operations: CO makes it all black, while OC makes it all white!

```
1 1 1 1 1 1
1 1 1 1 1 1
1 1 0 0 1 1
1 1 0 0 1 1
1 1 1 1 1 1
1 1 1 1 1 1
```

To investigate the advantages of word-shifting over serial bit operations, we'll use the notation $P \uparrow k$ [or $P \uparrow k$] to mean the bit vector P shifted left [or right] by k. We have $P^{\rightarrow_1} = P \vee (P \uparrow 1) \vee (P \uparrow 2)$ (RTB's earlier suggestion) and $Q^{\leftarrow_0} =$ Q A (Q $\uparrow$1) A (Q $\uparrow$2). But this conceals the fact that the P's are bit vectors that extend across word boundaries, so we should really write $P = (x, y, z)$ and $P^{\rightarrow_1} = (x \vee (x \uparrow 1) \vee (x \uparrow 2), (x \uparrow 35) \vee (x \uparrow 34) \vee y \vee (y \uparrow 1) \vee (y \uparrow 2), (y \uparrow 35) \vee (y \uparrow 34) \vee z \vee (z \uparrow 1))$. MES suggested that we could avoid shifts as long as 35 by masking out the bit we want, in case long shifts are slow.

The operation $P^{\downarrow_1}$ is, of course, easier since we work with 36-bit words and no shifting is needed,

DMO explained that he and AMY had handled the boundary condition by designating certain parts of the words (generally the bits at either end) as "trouble zones," then picking out enough bits to process the trouble zones using the simple algorithm. It turned out they needed exactly 36 bits to do this, so they could process the left and right halves of the picture, then put the boundary bits calculated for the trouble zones back in.

Continuing our algebraic manipulations, we wrote

$$P^{\rightarrow_1\leftarrow_0} = (P \vee (P \uparrow 1) \vee (P \uparrow 2)) \wedge ((P \uparrow 1) \vee P \vee (P \uparrow 1)) \wedge ((P \uparrow 2) \vee (P \uparrow 1) \vee P).$$

This formula involves eight logical operations, but we can reduce it to five by applying the distributive law:

$$P^{\rightarrow 1 \leftarrow 0} = P \wedge (((P \uparrow 1) \wedge (P \uparrow 2 \vee P \uparrow 1)) \vee (P \uparrow 1 \wedge P \uparrow 2)).$$

Such simplifications of formulas are usually possible; for example, DEK remarked that most **4-variable** formulas can be computed with five or fewer binary operations, and they all can be computed with at most seven, "Exclusive OR" is often helpful, but not for this particular function.

SOY presented a method for calculating P^O without shifting. Imagine 25 targets in each pixel, and a rifle with 25 bullets in each black pixel. We fire all the rifles, one bullet into each target that's in the 5×5 square of pixels surrounding the rifle that fires it. Then, we check each pixel to see if one of the nine 3×3 squares in which it's contained is all black. This method, which essentially calculates P^O by using relation **(2),** requires roughly $10n^2 + 25B$ basic operations and $25n^2$ pixels of space, where B is the number of black pixels. It demonstrates the value of our theoretical approach: via the nonobvious formula $P^O = P^{-\prec\succ}$ and **factorizations** like $P^{\prec} = P^{\rightarrow\downarrow}$, *we* are able to calculate P^O with $8n^2$ logical operations and n^2 pixels of space.

DCH explained a method for tracing the edges of the picture using only one raster. His method gives segments of the edges, not continuous cycles that can be traced. It has the advantage that it checks only around **1/9** of the raster points on average: it processes the picture in bands of three pixels, checking every third bit of one **row.** When it notes a color change, it knows that one or more boundary edges are needed, so **it** backtracks and puts them in, This algorithm relies on the cleanness of the pictures it's tracing.

JJF remarked that there is trouble with this method if we wish to know how the king moves hook together into a figure. DCH (and **ARR** and DAS) keep this information implicitly in the DD buffer, so they really do use an extra raster. **JJF** observed that the hooking together is a more global task, hence apparently less amenable to such a local approach.

Notes on Solutions to Problem 2

The writeups this time made delightful reading. Most of you wrote up your successful and **failed** approaches with a vengeance: the average writeup exceeded twenty pages! But you did a lot of interesting things on your respective paths to solution. Keep up the good work.

The passive voice is used by some of you quite a lot in your writing. An air of uncertainty is produced in your exposition by it. Since it is usually avoided by good prose **writers,** it ought to be avoided by you too, Since it appears so much in mathematical prose (**necessarily** so), it certainly ought to be eschewed by you at other times.

Your programs were, without exception, very nice to read. Nicely structured comments made even the most complicated ones reasonably easy to follow. The only stylistic comment is that some of you indent blocks too much (eight spaces or more): this makes deeply nested structures look pretty weird.

On the other hand, your programs were, almost without exception, not very nice to run. There were minor irritants, like programs that did not prompt the user for the name of the data file, (The grader **munged** those files when he could.) This is irritating in part because the **IOSAIL** macro **OPENREAD** (which **everyone used)** will prompt the user automatically **if you** don't provide a file name when you call it,

The big problem is that your programs really didn't interact very nicely with the user. Almost **no** one gave instructions, and the instructions were often ambiguous or unclear. It would have been nice, for example, to tell the user when a picture was being queued for output on the XGP; nicer still to ask whether the **user wanted it** output there; and, nicest of all, not to switch the display before asking the user on his other screen to make the decision!

As an example of the wise use of displays, consider the fact that several of your programs exploded during the king-move tracing phase. It would have been nice to switch the user back to his main screen so he could watch the execution time mounting or the error messages produced by your programs filling the screen. As it was, he had to guess whether the long waits were due to the system load or to bugs in your programs.

Only half of you **labelled** the figures you displayed. One nice feature no one had was a label on the display telling who had written the program that produced it. Such a line would have saved the grader a lot of guessing as to which programs produced which pictures in the mountain that collected the evening he ran all your programs.

All but one team cleaned the picture using the smearing **DRF** described, although the variety of descriptions for this method was astonishing. Almost everyone processed the king moves by creating an array of boundary points, then figuring out how to trace through the black pixels in that array.

RTB and AFB included an excellent description of the approaches they tried. This included a huge analysis of the cases that arise in edge-tracing. They also gave a *very* high-level description of their algorithm that made understanding their code a whole lot easier. They apparently found work on this problem to be almost a religious experience: after ten pages describing their method for cleaning **the** picture, they say, "Guilt abated and souls uplifted we now consider the edge tracing phase." One of the most heartening of your comments appears in their writeup: "We think **our** algorithm works quite well."

JEB started his writeup with a nice description of his progress through the **"theorems" suggested in** the problem statement, including the honest admission that

(5) $P^O = P^{-\prec-\succ}$ "struck me as unlikely.. . so I went on the the next 'theorem.'" His first work on the edge-tracing phase was to prove that cycles of boundary edges exist. He also analyzed quite carefully what information he'd need to draw the boundary, eventually developing a method that requires only one array. His discussion on storing the data in as compressed form as possible was also interesting.

DCH, ARR and DAS used a sound theoretical formulation and a careful examination of boundary space requirements to decide on their method. They reported trouble with SAIL's macro expansion facility that led them to code certain routines differently from their original plans. One nice idea they incorporated into their program was to adjust screen dimensions so that centers of 3 × 3 squares were at multiples of 7 in both directions. Their search for edges used absolute rather than relative directions, and gave pieces of boundaries rather than a continuous trace of the figures' edges; it managed to do this by examining only around 1/9 of the pixels in the raster.

JJF and MA1 also considered carefully what border pixels they'd need to clean the picture correctly. Their discussion of developing the edge-tracing algorithm is beautiful, especially considering their observation: "one of the real problems in listing 'bad algorithms' and/or 'bad ideas' is that quite a few of them are more like mystic revelations than algorithms; like a hologram they disappear as soon as you try to get your hands on them." They admitted, however, that the false starts helped them see what the important questions are. Eventually, they developed a theory of solid regions and interior points that led to an algorithm. They also included a discussion of options for storage management. Their writeup also contained some golden words: "The time that was well spent on this problem was spent away from any and all terminals, thinking rather than hacking."

DRF and MH had already described their method in class. Their writeup and code were the briefest *of* any of your solutions, Frankly, without having seen other peoples' code, the grader wouldn't have understood much of their program at all.

WDG and DRR established a time table for completion of various phases of their solution. Although this might mean their solution couldn't incorporate the latest ideas presented in class, it did have the advantage that they didn't have to rush any part of their work; it also facilitated their maintaining more regular hours. Their program included facilities for displaying their output either on a Data Disc or a Data Media, so they could accomplish much of their work from the terminal room in Polya. They also reported trouble with SAIL's macro facility, which forced them to write a logically boolean procedure as an integer procedure instead.

MSK and NCR were the only ones who didn't pack their data into integers. They used two arrays, stored the original picture in one, the closure in the second, then the clean picture back in the first; they worked straight from the definitions. They divided the work of writing the program cleanly between themselves, and

had only one problem when putting the pieces together: they had chosen exactly opposite directions for their coordinate system! They too reported trouble with SAIL, this time with its linking to an externally compiled procedure, Another of the heartening comments appears in their writeup: "I must admit to feeling very good about actually proving an algorithm before implementing it, and having the implementation work the first time. It is a rather vast improvement over my normal practices."

DMO and AMY tried using one of the tricks DEK mentioned once in class: transforming a problem so that it's easy, then transforming the answer back to the original. (Does $h = f^{-1}gf$ ring a bell?) Thus, they took out the bits that would cause trouble with a simple closure or interior routine and put them into a raster chosen so that they could be calculated with a non-complicated method. For the edge-tracing routines, it was clear that they liked using directions relative to the way the tracing element faces, They raised an interesting question in their writeup, too: What properties of a function f make it better t o compute in bit-serial fashion than bit-parallel?

CPP, CHT and WOL expressed their approach almost exactly as DRF had done. The dualism between closure and interior that they pointed out made the grader wonder why they had separate interior and closure routines. Their explanation of their edge-tracing algorithm was the clearest: they use the metaphor of bugs running around the boundaries of figures in the picture and being squashed together when they meet at a point. (Anyone using RAID or DDT on their program should be careful not to kill these bugs.)

MES was quite taken with the idea of developing a theory: he devised two new operations on pictures in his effort to axiomatize our algebra simply. He also did a very careful analysis of which operations could easily be performed in place in the raster. His program accepts any file as input, reading zeros and ohs as whites and all other characters as black; he made the intriguing observation that his program could be used to display the block structure of a program, He also managed to incorporate the dualism between interior and closure by writing *a* huge macro with parameters that he invoked almost like a procedure. His three tries at an edge-tracing algorithm led him to try developing a theory of "cores" *of* pictures. This task took him so long, though, that he didn't get a program for edge-tracing written!

IAZ included a nice discussion of how he decided what borders of the array he would need to compute the clean picture properly. He originally misunderstood the problem statement, so he wound up rewriting part of his code. As he pointed out, however, "this gave me a chance to be more consistent in naming conventions." His code also made the grader wonder about incorporating dualism in these programs; he's beginning to believe that on such a nitty-gritty level as bits the explicit coding of various cases is necessary,

initial picture p. type return to continue, p to print

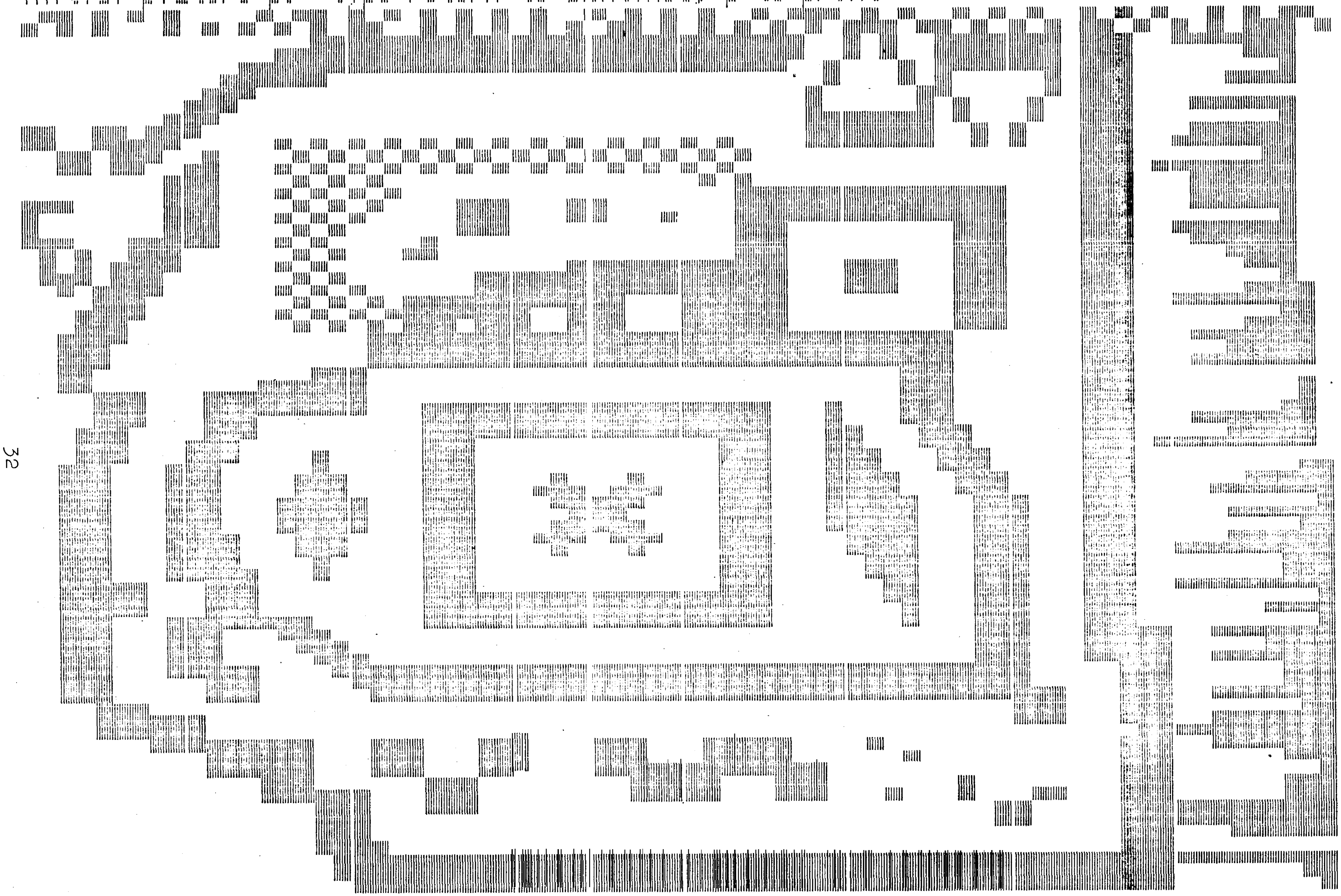
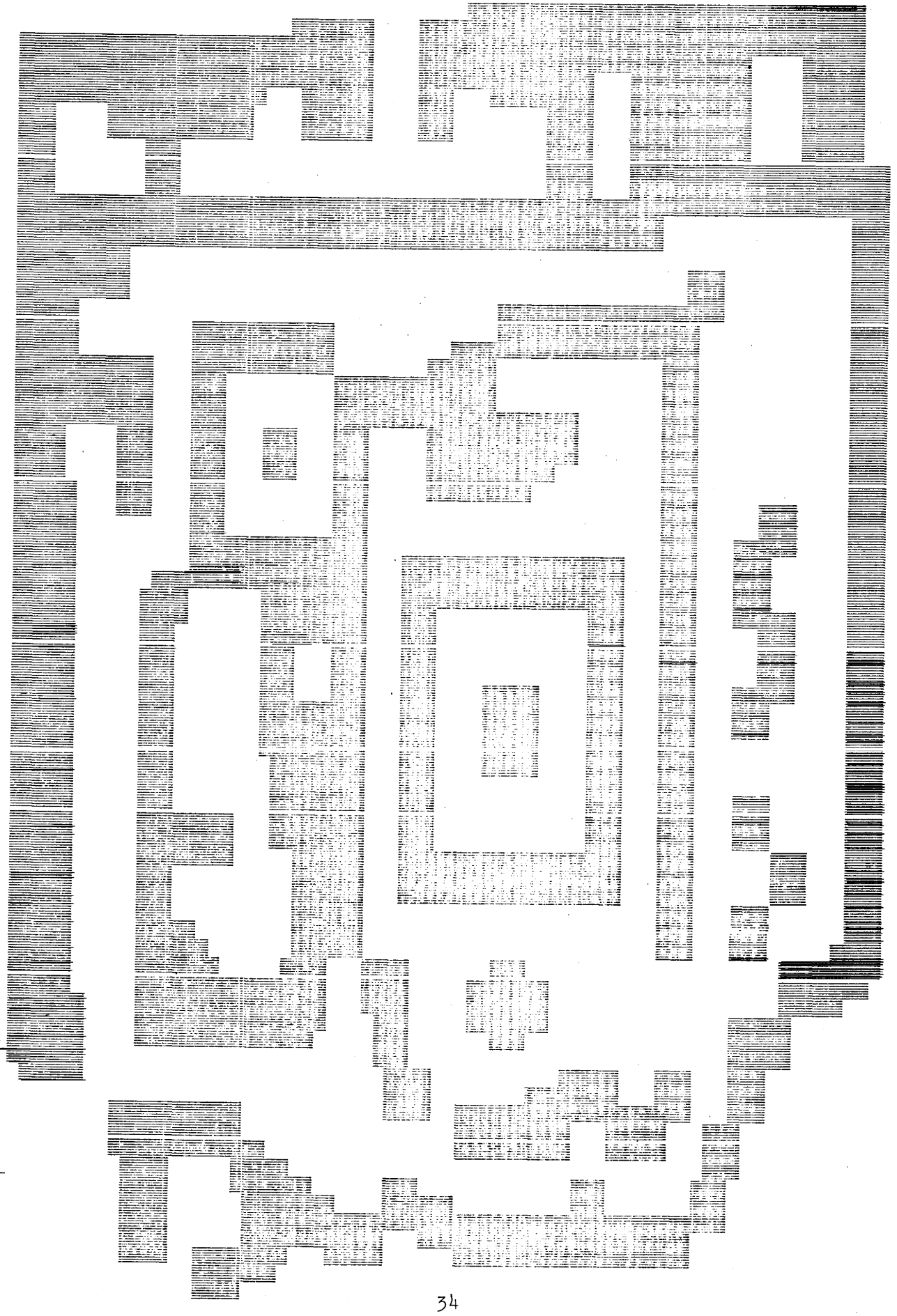
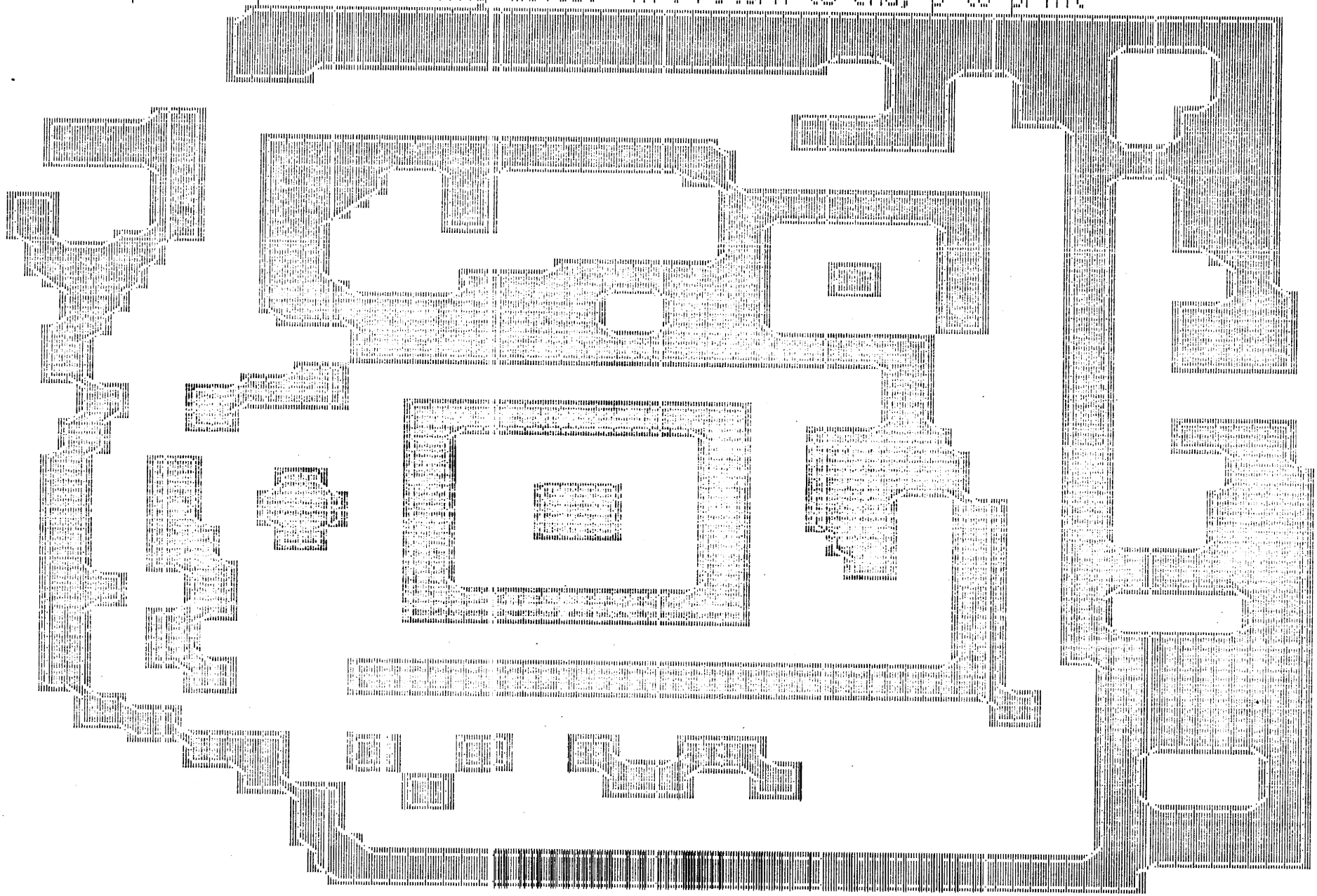


Figure 8. Fit feature to continuous print



cleared picture p with the king moves. hit return to end, p to print



The data used to test your programs is shown on the accompanying page. It is designed to give them a thorough workout: DEK called it " the dirtiest picture I've ever seen in my life." It starts out a picture within a picture within a picture within a picture, including multiply connected regions and some random patterns lying around. The rightmost of the figures inside the irregular box at center should merge with the frame around it, but the leftmost should stay separate. The checkerboard should fill in all black. The top left hand corner should be cleaned to all white. The grader's initials appear on the left border. The right border is a set of segments of random lengths.

Several teams had trouble with the top left border. Three programs went into apparently infinite loops trying to trace the king moves around the boundaries of the clean pictures. The pictures included here were produced by the program of **DCH, ARR** and DAS.

Problem 3. Cubic spline drawing (due November 2)

This problem arises in connection with drawing "nice" curves on a discrete raster. We *are* given two cubic polynomials

$$x(t) = x_0 + x_1 t + x_2 t^2 + x_3 t^3,$$
$$y(t) = y_0 + y_1 t + y_2 t^2 + y_3 t^3;$$

and we wish to plot the curve $(x(t), y(t))$ for $0 \le t \le 1$. But we are allowed to plot only integer points, making king moves.

Here is the way Knuth solved this problem in the prototype version of his "METAFONT" system last year. First, he made the problem slightly more general, namely to plot a set of curve segments $(x(t), y(t))$ for $a_i \le t \le b_i$ and for $1 \le i \le n$. Furthermore, it is possible to assume that $\frac{1}{2}$ has been added to x_0 and to y_0, so that the closest integer point to $(x(t), y(t))$ is $(\lfloor x(t) \rfloor, \lfloor y(t) \rfloor)$. Under these conditions, the algorithm repeatedly does this: Terminate if $n = 0$. Otherwise, if $x(t)$ is not monotonic between $t =$ a, and $t = b_n$, find the roots of its derivative $x'(t)$ and break the interval into two or three intervals in which $x(t)$ is monotonic (increasing n accordingly). Similarly, the intervals are broken up even further, if necessary, so that both $x(t)$ and $y(t)$ are monotonic between $t = a_n$ and $t = b_n$. *If* now $\lfloor x(a_n) \rfloor = \lfloor x(b_n) \rfloor$ or $\lfloor y(a_n) \rfloor = \lfloor y(b_n) \rfloor$, draw the appropriate horizontal or vertical straight line (a rook move); decrease n by 1. Otherwise if $\lfloor x(a_n) \rfloor = \lfloor x(b_n) \rfloor \pm 1$ and $\lfloor y(a_n) \rfloor = \lfloor y(b_n) \rfloor \pm 1$, make the appropriate diagonal king move; decrease n by 1. Otherwise replace the interval [a,, b_n] by two intervals [a,, $\frac{1}{2}(a_n + b_n)$] and $[\frac{1}{2}(a_n + b_n), b_n]$; increase n by 1.

Knuth's "binary splitting" method seems somewhat elegant at first glance, but the curves it draws are sometimes too skinny-looking and they occasionally contain undesirable glitches near points where t is a simple binary fraction, Even when x(t) and $y(t)$ are linear, the results are often *very* strange, So the purpose of this problem is to help Knuth design a better routine for the real METAFONT system. The new approach is intended to be faster and to give better curves.

Here's the germ of an idea that should work better: By refining the intervals a little more, we can ensure that $x(t)$ and $y(t)$ are not only monotonic in each interval, but that their slope $y'(t)/x'(t)$ is never equal to ± 1 (unless the slope is constant in the interval). This means that the curves in each interval will now be confined to one of eight "octants"; by rotation and/or reflection we can assume, for example,

that $x(t)$ and $y(t)$ are increasing and that $y(t)$ increases less rapidly than $x(t)$. (In other words, $0 \le y'(t) \le x'(t)$.) Now each move is either one rectilinear step to the right, or one diagonal **step** up and right.

Find an efficient way to plot a given cubic **spline** curve $(x(t), y(t))$ for $0 \le t \le 1$, assuming that $0 \le y'(t) \le x'(t)$ for $0 \le t \le 1$. Your algorithm should decide as rapidly as possible whether or not each successive rightward move should have slope 0 or 1. Then incorporate your method into an efficient subroutine for the general problem. [Hint: We would like to plot the points $(n, \lfloor y(t_n) \rfloor)$, where $x(t_n) = n$; but it is probably not necessary to know the value of t_n very accurately.]

Notes from October 24

DEK was sick today, so CVW led the class, This had the potential advantage that since CVW is not as familiar with cubic splines as DEK is he would be less likely to interject his own ideas into the discussion, It has the possible disadvantage that what DEK had in mind when he wrote the problem is not known to CVW any more than to anyone else.

Several people wanted to know what DEK meant by "the curves it [his binary splitting method] draws are sometimes too skinny-looking and they occasionally contain undesirable glitches near points where t is a simple binary fraction." DRM suggested that some undesirable skinniness arises when a diagonal line is drawn and it appears thinner by a factor of $\sqrt{2}$ than a line parallel to one of the coordinate axes. JJF and DRF both suggested that problems could arise because of insufficient resolution in the raster, JEB posited that glitches near binary fractions could arise when drawing a straight line and two jumps between pixel rows occurred close together, instead of being fairly evenly distributed over the length of the line. No one had done enough playing around with splines, though, to give an example of a funny-looking raster curve produced by DEK's binary splitting method.

A couple of people who had heard DEK's talk on mathematical typography (see Mathematical Typography, STAN-CS-78648, February 1978) thought that the problem might have to do with what shape pen tip was used to trace around the spline curve on the raster. Again, this discussion was inconclusive, because none of us can read DEK's mind. NCR suggested that we use some statistical approach to evaluate methods of plotting splines, but the problem statement suggests that there's an esthetic component to the judgment of the results too.

Disregarding what DEK wants, for the moment, we discussed several methods that were presented for plotting splines. We even restricted the discussion to the special case he mentioned at the end of the problem statement: plot a given cubic spline curve $(x(t), y(t))$ for $0 \le t \le 1$, assuming that $0 \le y'(t) \le x'(t)$ for $0 \le t \le 1$. We can try to beat the speed of DEK's binary splitting method, or we can establish some measure that compares our curves with those produced by binary splitting.

ML followed up on the hint given at the end of the problem with this approach: Suppose we wish to plot the points $(n, y(x^{-1}(n)))$ *for* $n = \lfloor x(0) \rfloor, \lfloor x(0) \rfloor + 1, \ldots, \lfloor x(1) \rfloor$. We can use a Newton-Raphson approximation to find *a* point t_n such that $x(t_n) \approx n$. Suppose we have previously plotted (n-l, k); then the new point will be either (n, k) or $(n, k+1)$, and the problem is to decide which. If $x(t_n) \ge n$ and $y(t_n) < k+1$, the answer is (n, k). If $x(t_n) < n$ and $y(t_n) \ge k+1$, the answer is $(n, k+1)$. Otherwise, we need a better approximation. This idea is based on the hope that our rough guess to t_n will be good enough to tell the general direction in which the curve is going without much computational labor.

CHT suggested that instead of a Newton-Raphson approximation we compute the first few terms of a Taylor series for t_n. He also observed that DEK's binary splitting method does at most four function evaluations at each step of the algorithm, which suggests that we do want to restrict our computation rather heavily, It also mandates a rather careful error analysis before we program the computation, since we want it to be robust for a variety of spline curves.

MES asked just how little computation **DEK's** binary splitting does. The way it's written in the problem statement suggests that a derivative is evaluated at each step. After some discussion, we decided that the roots of the derivative of $x(t)$ are found before the splitting starts, and then we can check quickly whether either or both lie in an interval we're processing. In the latter case, the interval is split up into two or three separate intervals.

JEB showed us what he called "the **state-of-the-art**" algorithm for drawing straight lines on a raster. Say the (positive) slope of the line is dy/dx. We put dx into a register and repeatedly subtract dy from it. After each subtraction, we check the sign of the register; if positive, make a rightward horizontal move; if negative, make a diagonal move upwards and add dx to the register. The existence of such a nice method for lines prompted CPP and **DRF** to suggest that we break the spline into pieces that can be approximated by lines, then use this algorithm on each of them.

Notes **from** October 26

DEK was still on sick leave today, We spent most of class discussing improve ments to the basic algorithm ML showed us last time, Recall that his method was to start at $t = 0$, and use $t_{11} = 1/x'(0)$ as a first approximation to t_1. If $x(t_{11}) > 1$ and $y(t_{11}) < \lfloor y(0) \rfloor + 1$ then we move straight to the right; if $x(t_{11}) < 1$ and $y(t_{11}) > \lfloor y(0) \rfloor + 1$ then we move diagonally upward, If neither inequality holds, obtain a better approximation t_{12} to t_1 and repeat the above tests for which move to make.

CPP observed that if we could derive an approximation formula for t_n for which $x(t_n)$ lay alternately on one side or the other of n (**viz.**, $x(t_{n,2k}) <$ n and $x(t_{n,2k+1}) > n$ or vice versa), we'd be in good shape: this method probably wouldn't require too many iterations before it computed a good enough approximation to t_n so that we could make our move and go on,

We discussed the possibility that the Taylor series approximation to t_n is alternating, hence has the nice behavior we'd like. Although it's not true in general that a Taylor series alternates (the polynomial $x^3 + x^2 +$ x + 1 being its own Taylor series, for example), we might hope for better luck in our situation, because inverting $x = 1 + t^3$ gives an alternating Taylor series for t in terms of x. But what about $x = 1 + t^2 + t^3$? The class was pretty evenly divided between those who believed the Taylor series inversion would alternate for most **cubics** and those who weren't

convinced.

ARR suggested that we try $t_{12} = \frac{1}{x(t_{11})x'(0)}$ if we needed a better approximation to t_1. This looks sort of like a second-order Taylor series approximation, but ML pointed out that it's cheap to compute: since we already have $1/x'(0)$ and $x(t_{11})$, *we* need only perform one division.

WDG proposed that if we still needed a better approximation to t_1 after two tries, one low and one high, we use the abscissa of the intersection of the tangent lines at the *two* guesses to get a very good guess. The formula for this,

$$x = \frac{1}{x'(t_1) - x'(t_2)}\left(x(t_2) - x(t_1) - t_2x'(t_2) + t_1x'(t_1)\right),$$

involves computing only one value we won't already know, $x'(t_2)$, which is probably not too much work.

DRM observed that these approaches all fit the following general pattern: we know two numbers s_1 and s_2 such that

$$x(s_1) < \hat{x} < x(s_2)$$
$$y(s_1) < \hat{y} < y(s_2)'$$

where $(\hat{x}, \hat{y})$ is "special" (for example, $\hat{x} = 1$ and $\hat{y} = \lfloor y(0) \rfloor + 1$). Then any s_3 we choose in the range $s_1 < s_3 < s_2$ can either replace one of s_1 and s_2 (because it satisfies the same inequalities) or gives enough information to decide where to move. Our basic problem is deciding how to pick s_3: using the midpoint gives binary splitting.

DCH suggested simply writing out the Taylor expansion of t to third-order: $t(x) = a_0 + a_1x + a_2x^2 + a_3x^3$, then plotting the moves determined by the set $\{t(i), y(t(i))\}_{0 \le i \le n}$. WOL observed that this can't work if the interval is very long, since $t(x)$ is a good approximation only in a neighborhood of x = 0. But it might be a reasonable idea to use the same approximation formula for a few steps across the raster, then adjust the formula by computing the Taylor expansion about a different point.

NCR asked if there was a standard spline curve to use for testing programs. DRM suggested that $x(t) = y(t) = (t - 1/2)^3$ might be interesting to try: it's a terrible **parameterization** of a straight line,

IAZ proposed a method by which he hoped to reduce the problem to solving a quadratic equation. We'd like to know where the curve is with respect to the point (n, m), m being the row we crossed on the y-raster when we plotted the point $(n - 1, m)$. One way to try to find this is to intersect a line through (n, m) of slope y_3/x_3 with the curve. This gives us a quadratic in t

$$x_3(y(t) - m) = y_3(x(t) - n)$$

(because the t^3 terms cancel), which we can solve to find an approximate value for t_n.

NCR suggested that we use some method based on the spline's curvature to take longer steps in drawing the curve. Along these lines, DAS observed that parts of the spline that were troublesome to fit with a straight line might be approximated well by a parabola.

Notes from October 31

DEK started class asking who had sent him an article from IBM's Technical *Disclosure Bulletin* (September 1978) about drawing straight lines on bit rasters. The article described yet another "discovery" of the method JEB showed last week, which DEK thinks was first published in 1965 [IBM Sys. *J.* 4, 25].

DEK showed us the test case he had in mind when he wrote up Problem 3, his "favorite" cubic splints. Let

$$z(t) = z_0 + (z_1 - z_0)(3t^2 - 2t^3 + re^{i\theta}t(1 - t^2) + se^{i\phi}t^2(1 - t)).$$

Then $x(t) = \Re(z(t))$ and $y(t) = \Im(z(t))$ is a good test case. This is the most general cubic from z_0 to z_1 that starts at angle θ relative to the straight line $[z_0, z_1]$ and comes in at angle ϕ. The parameters $r > 0$ and $s > 0$ determine how close the spline stays to the line $z(t) = z_0 + (z_1 - z_0)t$. For $\theta = \phi$, we get a cubic spline that looks very close to an arc of a circle, if we choose r and s so that the spline hits the circle at $t = 1/2$. In general, DEK's METAFONT system chooses r and s as functions of θ and ϕ as follows

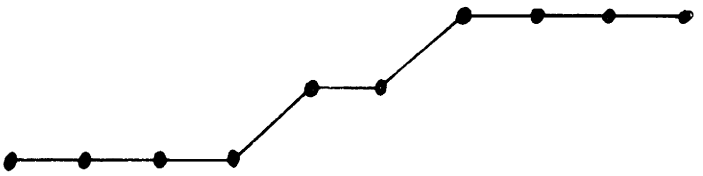
$$r = \max\left(\frac{1}{2}, \left|\frac{4\sin\phi}{\left(1+\cos\frac{\theta+\phi}{2}\right)\sin\frac{\theta+\phi}{2}}\right|\right)$$

$$s = \max\left(\frac{1}{2}, \left|\frac{4\sin\theta}{\left(1+\cos\frac{\theta+\phi}{2}\right)\sin\frac{\theta+\phi}{2}}\right|\right).$$

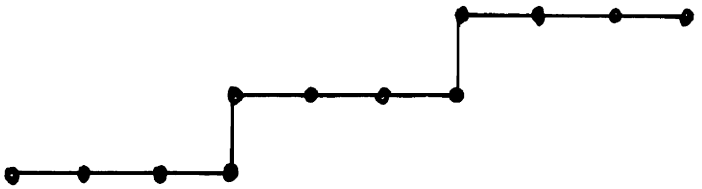
He suggested we test our programs on a family of cubics with $z_0 = 0$, $z_1 = 1$, fixed θ but varying ϕ.

In response to NCR's question, DEK clarified the problem statement to the effect that $x(t)$ and $y(t)$ need not satisfy the derivative condition $0 \le y'(t) \le x'(t)$ over their whole domain of definition: instead, we'd split the domain into pieces so that the condition was met by the two equations, possibly interchanging the roles of x and y.

Here's what DEK meant when he referred to "skinny lines":



looks skinnier than



when the points are replaced by black square **pixels.** He observed that needlepoint samplers **usually draw straight lines using the second** form, whereas **binary splitting** produces a mixture of **skinny** and fat. Two ways to compensate for the skinniness problem are to **use** a thicker pen and to try to fill in one of the "corner" squares in the raster. So for Problem *3* it's OK to use the skinniest curve.

In reply to a question from JJF whether this is a serious problem at a resolution of *5000* pixels per inch, DEK stated that he wants his spline plotter to be as independent of the raster resolution as possible: the fonts produced by the program should not require editting by hand to repair glitches when the resolution is, say, 100 per inch or more. CHC asked if one *ever* needed lines in fonts that are just one pixel wide, Answer: Yes, "hairlines" and serifs are typically one pixel wide. DEK called the exclamation point the biggest problem, since it should taper in width from top to bottom, but it's not easy to decide on a raster where to go from width 4 to width 1 and make the thing look centered.

Next, DEK presented a method that no one had mentioned in discussion so far, namely eliminating dependence on t. Say

$$x(t) = a_3t^3 + a_2t^2 + a_1t + a_0$$
$$y(t) = b_3t^3 + b_2t^2 + b_1t + b_0.$$

If we write

$$\sum_{0\le i<3} a_it^i - x - c \sum_{0\le i<3} b_it^i - y = 0,$$

we can compute the resultant of those two polynomials, alias Sylvester's determinant:

$$F(x,y) = \det \begin{pmatrix} a_3 & a_2 & a_1 & a_0 - x & 0 & 0 \\ 0 & a_3 & a_2 & a_1 & a_0 - x & 0 \\ 0 & 0 & a_3 & a_2 & a_1 & a_0 - x \\ b_3 & b_2 & b_1 & b_0 - y & 0 & 0 \\ 0 & b_3 & b_2 & b_1 & b_0 - y & 0 \\ 0 & 0 & b_3 & b_2 & b_1 & b_0 - y \end{pmatrix}$$

Then $F(x, y) = 0$ is a third degree polynomial in x and y; in general,

$$F(x, y) = ax^3 + bx^2y + cxy^2 + dy^3 + ex^2 + fxy + gy^2 + hx + iy + j.$$

Now, we can express our rule for moving across the raster this way:

```
while appropriate do
   begin
   x + x + 1 ;
   if F(x, y) < 0
      then begin
           y ← y + 1;
           move diagonally
           end
      else move right
   end
```

An advantage of this is that the test $F(x, y) < 0$ and the updating of x and y can be done using only six registers. This makes it a nice candidate for implementation by microprocessor. We use a difference method to perform the computation. We'll only need six registers because each (partial) differentiation reduces the degree of the polynomial $F(x, y)$ by one.

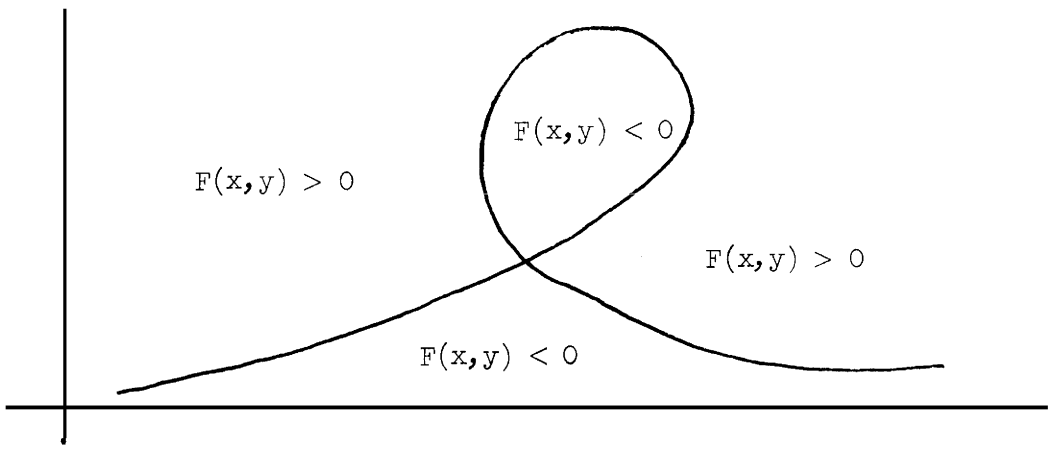
Here's how the algorithm looks with the difference computations:

```
while appropriate do
   begin
   [x ← x + 1]
   F ← F + Fx;
   Fx ← Fx + Fxx;
   Fxx ← Fxx + Fxxx;
   Fy ← Fy + Fyx;
   Fyx ← Fyx + Fyxx;
   Fyy ← Fyy + Fyyx;
   if F < 0
      then begin
           [y ← y + 1]
```

$F \leftarrow F + F_y$;
$F_y \leftarrow F_y + F_{yy}$;
$F_{yy} \leftarrow F_{yy} + F_{yyy}$;
$F_x \leftarrow F_x + F_{xy}$;
$F_{yx} \leftarrow F_{yx} + F_{yxy}$;
$F_{xx} \leftarrow F_{xx} + F_{xxy}$;
move diagonally
end
else move right
end

Here, $F_x(x, y) = F(x + 1, y) - F(x, y)$, $F_y(x, y) = F(x, y + 1) - F(x, y)$, $F_{xy}(x, y) = F_x(x, y + 1) - F_x(x, y)$, etc. The values of F_{xxx}, F_{xxy}, F_{xyy} and F_{yyy} are constant. The derivative condition $0 \leq 3/(t) \leq d(t)$ is important or this would not work, Perhaps other conditions are needed too: we have eliminated t_1 and the equation $F(x, y) = 0$ includes points not on our original curve.

JEB asked whether this method will have trouble if the cubic spline intersects itself. Problems arise because we're using the sign of $F(x, y)$ to decide which direction to move, but the side of the curve on which $F(x, y)$ is negative suddenly switches when the curve intersects itself:



DRM asked whether the six-register method could be slightly modified so that a change *of* octant is detected as soon as it occurs; then we wouldn't need to preprocess the curve to break it into pieces. DEK said he didn't know if this method can be fixed up easily, but its efficiency makes it an appealing direction to pursue, However, he didn't mean to say that it was the only promising line of attack; he liked the methods discussed in class last week too,

DEK suggested another possible algorithm: take big steps across the raster, rather than asking at each raster column which direction to go. Such a method could be better than linear in processing time.

In response to the wailing and gnashing of teeth about the ailing SU-AI computer, DEK suggested that this is a golden opportunity to consider our theory very carefully before trying to implement it.

Notes from November 2

We continued to discuss ideas about cubic spline plotting. Because of SU-AI's temporary difficulties with its disk channel, few people had much experience with programs running on actual curves. There were plenty of ideas anyhow.

NCR elaborated on the idea of breaking the spline curves into pieces short enough that each could be approximated well by a straight line. First, he finds where dy/dx or dx/dy vanishes. Since $\frac{dy}{dx} = \frac{dy/dt}{dx/dt}$, we can find where dy/dx vanishes by solving the quadratic equation $dy/dt = 0$; a similar observation applies to the places where dx/dy vanishes. So there are at most four interesting points of this kind. If we also designate the endpoints of the interval to be interesting, we have at most six interesting points at $t = s_0, s_1, \ldots, s_5$.

Next draw straight lines between successive interesting points (that is, between $(x(s_i), y(s_i))$ and $(x(s_{i+1}), y(s_{i+1}))$ for $0 \leq i \leq 5$), then smooth the resulting polygonal curve so as to reduce the angles between successive segments. To do this latter smoothing, we replace the segment between $(x(s_j), y(s_j))$ and $(x(s_{j+1}), y(s_{j+1}))$ by two segments, one between $(x(s_j), y(s_j))$ and $(x(\frac{s_j+s_{j+1}}{2}), y(\frac{s_j+s_{j+1}}{2}))$, the other between $(x(\frac{s_j+s_{j+1}}{2}), y(\frac{s_j+s_{j+1}}{2}))$ and $(x(s_{j+1}), y(s_{j+1}))$.

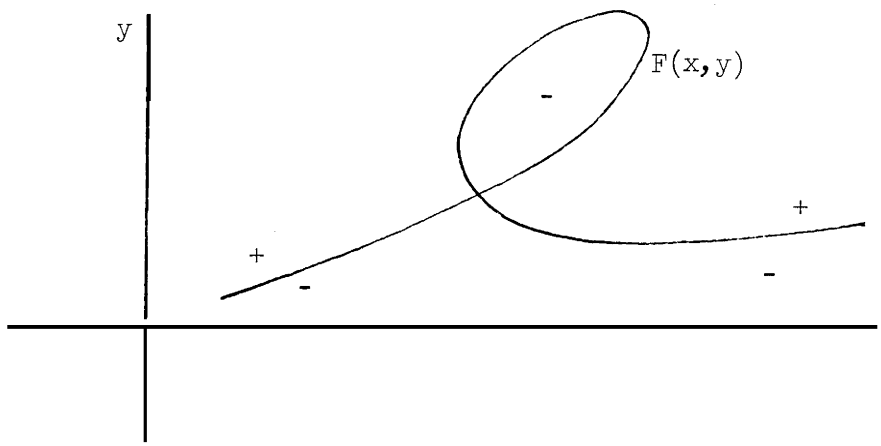
JEB asked if there would be a problem when both dy/dt and dx/dt vanish: such places are cusps on the curve, where neither dy/dx nor dx/dy exists,

IAZ observed that this problem will not arise if we allow such points only at the endpoints of our intervals, Furthermore, it's easy to insure that this condition holds: if we solve the quadratic equation

$$y_3x(t) - x_3y(t) = 0 \qquad (*)$$

for t, the discriminant turns out to be independent of x and y. The geometric significance of the discriminant is that it's zero when both derivatives dy/dt and dx/dt are equal, hence where both vanish *if* that happens. IAZ also noted that substituting the value we get for t out of (*) back into (*) and rationalizing the resulting equation gives us the same polynomial as Sylvester's determinant.

DEK was still worried, however, that the six-register method he presented last time would fail in a situation like that pictured below, because it relies on the polynomial $F(x, y)$ being negative below the curve and positive above, but the signs switch when the curve crosses itself. This looks like it'll give us trouble even though we're on one branch of t (discriminant always of same sign).

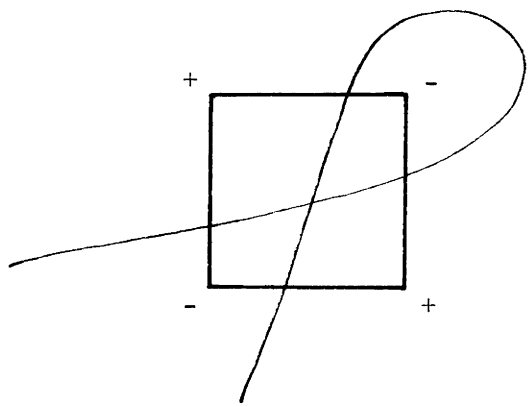


To remedy this problem, **JEB** suggested that we could look at F_y as well as at F when we decide in which direction to move. **DRM** observed that the curve crosses itself when the Jacobian of F vanishes:

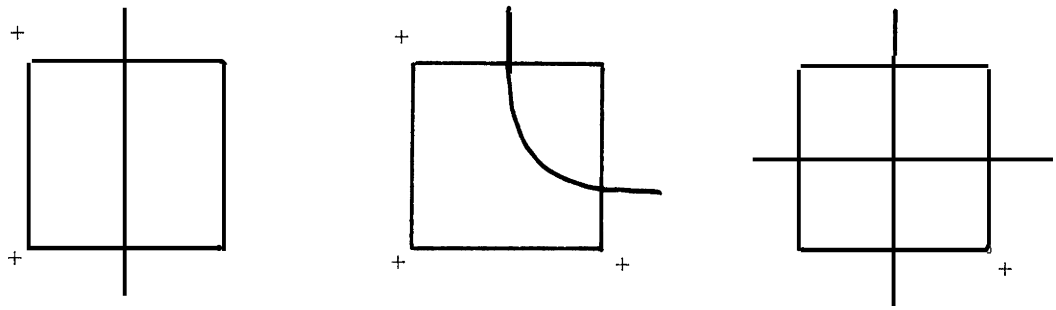
$$\det\begin{pmatrix} F_{xx} & F_{xy} \\ F_{yx} & F_{yy} \end{pmatrix} = 0.$$

The observation that the crossover is a saddle point doesn't seem to help in the plotting: we'll just have to look to see when the sign of F_y changes.

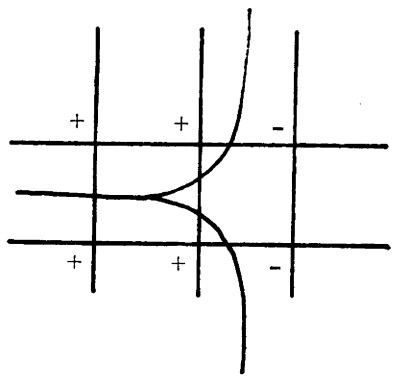
JEB asked about this case: (The signs tell the sign of $F(x, y)$ at that corner.)



This reminded DEK of algorithms for contour lines, which use the sign pattern to decide which of three possible lines to draw inside a raster square.



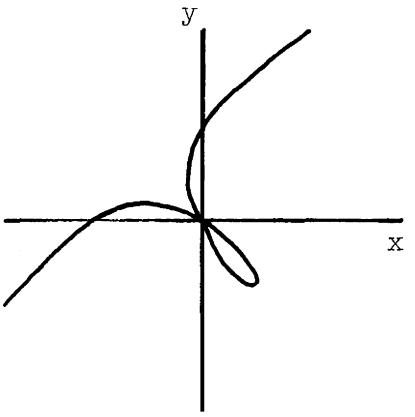
These algorithms also have the problem that they'll chop off curves like this one:



ARR pointed out that this might not be so bad, since such spikes would be cleaned out of pictures by the algorithms resulting *from* Problem 2 anyhow.

We discussed troublesome curves *for* about forty-five minutes. Finally, **DRM** showed **us** this example of a troublesome curve:

$$x(t) = (t+1)(t-1)(t-3)$$
$$y(t) = (t+3)(t+1)(t-1)$$



CHT presented his team's method next. Say the slope at t_0 is s_0. Solve the equation $\tan(|\tan^{-1} s_0| \pm \delta) = dy/dx$ *for* δ to within some tolerance. Then draw a line of the appropriate slope from (x(O), y(0)) to where it intersects the curve $(x(t), y(t))$. This method requires work proportional to the curvature of the spline.

DCH elaborated some on the method he showed us last Thursday. Recall that he was going to calculate the Taylor expansion of t(x) and use it to walk across the raster. Since Taylor approximations are generally valid only in small neighborhoods, he said last time he'd adjust his Taylor expansion every once in a while. He could do so either at certain pre-chosen points in the plotting or adjust whenever some tolerance was violated. He'd chosen the latter method; he observed that the approximation could be pretty far off (around .9 pixels) and he'd still be close to the intended curve, so such tolerance checks needn't be very costly.

DEK pointed out that since he'd specified that $0 \leq y'(t) \leq x'(t)$ in the interval we're plotting, it might be better to walk along the side of the raster, that is, to find the u_n at which $y(u_n) = n$. For example, the spline $x(t) = t$, $y(t) = .01t$, is easily drawn this way because the curve will go straight for 100 x-points before rising by one y-point.

ARR and MES asked whether the potentially huge error in computing u_n (because the function y is changing so slowly) might cause problems with this approach. But JEB observed that we can afford to do about 100 times as much computation to make our u_n values good as we'd do travelling in the x-direction. CHT noted that his team's method would do well on this type of curve, since it doesn't prefer one direction on the raster over the other.

MES questioned our having spent so much time on special cases like cusps and curves of small curvature. "'It seems," he said, "that each method that's been presented is optimal *for* some class of curves. Bow can we come up with a method

when we have no idea of the particulars of the curves which will confront it?" DEK remarked that this is the dilemma of computer science: there rarely is one method that's always superior to another for any input.

Notes on Solutions to Problem 3

RTB and AFB tried several methods (including a couple that ignore the octant in which the curve lies) before amalgamating the best features from each into their final solution. They break the interval into segments and approximate the curve on each segment by a Lagrange polynomial of degree at most two. Then they plot the Lagrange polynomial using binary splitting. This is faster than simple binary splitting because the function evaluation is cheaper. Their curves looked very nice, too. DMO and AMY's work proceeded along lines quite similar to RTB and AFB's. Their method uses Hermite cubic polynomial approximations rather than Lagrange quadratic polynomials, however,

JJF and MH implemented a simple method: they solve for the values of t at which $x(t)$ is an integer, using *regula falsi,* compute the value of $y(t)$ at these points, and plot (x, y).

DRF and MA1 devised a "pseudo-curvature" function that's easy to calculate and tells what set of straight lines to draw that will approximate the curve reasonably well. Their method handles cusps especially nicely.

WDG and DRR followed the hint given in the problem handout pretty directly, with the modification that they sometimes step along the y-axis rather than always stepping along the x-axis. They used bisection to find the t_n where $x(t_n) = n$ (or $y(t_n) = n$) because bisection is guaranteed to converge while simpler methods aren't. They also observed that the eight octants could be replaced by four quadrants separated by the lines $y' = \pm x'$; for example, the case $0 \le y' \le x'$ is like $0 \le -y' \le x'$.

CPP, CHT and WOL implemented a method that takes account of the curvature of the spline, drawing long straight segments very fast. They conceive of their algorithm as sliding a "window" (shaped like a sector of a circle) along the curve: with the window sitting on the curve, they check where the curve intersects one of its sides, and draw a straight line between the window's corner and that intersection. The speed of computation depends on the angle of the window. This method produced some very nice pictures.

IAZ's method uses knowledge about which octant the spline curve lies in. We state it here as if the curve lies in the first octant, If the point $(n - 1, m - 1)$ has been plotted, we must decide between drawing a straight line to $(n, m - 1)$ or a diagonal line to (n, m). To decide this, evaluate the implicit equation for the curve $(F(x, y) = 0)$ at (n, m); if the sign differs from what it was at $(n - 1, m - 1)$, move diagonally upward, else move straight to the right, Because the implicit equation

can be found in a straightforward fashion *(cf.* Sylvester's determinant) and is *easy* to evaluate, this method is quite fast.

NCR presented a nice discussion of a variety of possible approaches to the problem, though he hadn't been able to get very far implementing his ideas because of **SU-AI's** yo-yo mode while we worked on this problem. He attempted to quantify the notion of "skinniness" and use the skinniness properties of the pictures drawn by each method to decide which methods were superior, He also gave several examples of "glitches" and what can be done to resolve them, Some of his methods plotted point-by-point across the grid, while other6 made use of more global knowledge about the spline's curvature so as to take bigger steps across the raster between function evaluations.

MSK was also stymied in his efforts by the system's flakiness, He found it hard to come up with much in the absence of experience with curve-plotting, though. He suggested one interesting possibility in his writeup: refine **binary** splitting to work on smaller sector6 than **octants** (hexadecimants?, **duotrigimants?,** etc.).

MES handed in an excellent survey of his work on the problem, although he also had had trouble getting a program running in the face of **SU-AI's** troubles. He proposed a rule for moving that's very similar to the idea ML presented the first day we looked at this problem, with the proviso that if our root-finder for $x(t_n) = n$ finds t_n to a certain tolerance but no decision about which move to make has been made, we select one of the two possible moves randomly and proceed.

DAS presented *a* heroic amount of calculation devoted to **figuring** out a nice way to express the curve in an easily computable form that avoids the dependence on t. He gave some examples that show just how hard it is to approximate a semicircle by a univariate polynomial of low order,

DCH and **ARR** used a Taylor series approximation to solve $x(t_n) = n$ for t_n. They change the point about which the function is expanded in its Taylor series only when they find they've deviated significantly from the true curve.

Problem 4. A language for Gosper arithmetic (due November 16)

The main purpose of this problem is not to design a computer program-instead, we will try to design a good programming ***language of*** a new type (related somewhat to the so-called "data-flow" languages being discussed nowadays in connection with database systems). The semantics of a certain new computational scheme will be sketched below; your task will be to suggest a good interface by which a user will be able to use the scheme intelligently.

Here's the sketch: Every real number x can be represented in a unique way as a continued fraction

$$x_0 + \cfrac{1}{x_1 + \cfrac{1}{x_2 + \cdots}}$$

where x_0 is an integer and $x_1, x_2, \ldots$ are infinitely many positive integers; or perhaps $x_k = \infty$ for some k, in which case $x_{k+1}, x_{k+2}, \ldots$ do not exist, For convenience in notation, we write $x = x_0 + /x_1, x_2, \ldots /$. The x_k's are called "partial quotients" of x.

R. W. Gosper has developed a set of routines that do arithmetic directly on the continued-fraction representations of numbers. His routines have the neat property that, when computing something like

$$z_0 + /z_1, z_2, \ldots / \leftarrow (x_0 + /x_1, x_2, \ldots /) + (y_0 + /y_1, y_2, \ldots /),$$

they look at just enough *of* the x_i and y_j to determine z_k before z_k is output, for $k = 0, 1, 2, \ldots$. Thus it is best to think of his arithmetic procedures as ***coroutines*** that produce one partial quotient at a time when called on. Gosper's addition coroutine for $z \leftarrow x + y$ is effectively hooked up to the coroutine for x and the coroutine for y; when the z coroutine is asked for a new partial quotient, it will call on the x and/or y coroutines for more information, but only as much as necessary.

Actually every numeric quantity in the system is a coroutine, at least at the user's level of abstraction, and these coroufines are connected together in a possibly gigantic network, There are no ordinary numbers, in the old-fashioned static sense of some bit representation stored somewhere; numeric quantities are totally dynamic. Nothing is ever computed "to the end," the calculations just get more and more precise if this is called for.

When started or restarted, a coroutine for x will sooner or later produce an output that is either the next partial quotient for x or it will produce a coded message that means "x is indeterminate, but its next partial quotient is anything between a and b inclusive, and don't ask me for any more information." .**This** allows us to deal with inaccurate input data, producing exactly as much output precision as is legitimate.

Assume that the following types of coroutines are available:

(1) Given two decimal numbers x and ϵ, output the partial quotients for a number that is indeterminate but between x-c and x $+\epsilon$.
(2) Given two coroutines x and y, output the partial quotients for x + y.
(3) Given two coroutines x and y, output the partial quotients for x $-$ y.
(4) Given two coroutines x and y, output the partial quotients for x $\times$ y.
(5) Given two coroutines x and y, output the partial quotients for x/y.
(6) Given a coroutine x, output the decimal digits of x.

Each of these coroutines will produce its outputs one at a time, as called for. There is also a procedure that, given coroutines x and y, will produce one of four outputs: "$x < y$", "$x = y$", "$x > y$", or "indeterminate".

All this sounds very nice, but there is a serious problem that has been glossed over: In order to produce results in finite time, the coroutines sometimes have to make guesses and retract bad guesses later, Furthermore the comparison procedure might have to give a fifth output, "I can't figure it out even though I've checked lots of partial quotients of x and y." The reason can be understood by considering the analogous situation where we are working in the decimal number system instead of with continued fractions: After reading finitely many digits of $\sqrt{2} = 1.4142...$, we will not be able to tell if the square of this quantity is less than 2 or not, even though we know that it is at least 1.99999999999... (say); we have to wait forever before we know even the first digit of the product. When Gosper's routines appear to be computing too long, he makes them produce a guessed-at result; if this turns out to be wrong, they will later output a partial quotient that is zero or negative, in such a way that the continued fraction will still evaluate to the correct final value.

So that's the general idea. Our goal is to design a suitable user programming language. To make the problem more explicit, you are asked to illustrate how you would solve the following two problems using the language you design:

(a) Calculate a root of any given cubic equation.
(b) Solve a system of n simultaneous linear equations (by some straightforward scheme).

For each of these tasks, present a sample program in your new language. Also give a brief description of the features of your language. (But don't attempt to implement a compiler for it, wait until you take CS240 or something.)

Notes from November 7

A large part of this session consisted of DEK's preamble to Problem 4. He explained that an exciting new idea in programming is to view variable6 as dynamic rather than static entities: they can answer questions about themselves, The problem statement suggests one such view where the variables are numbers. A non-numeric example would be the merging of two sorted lists: if the elements of two lists can be specified by some rules, we can imagine merging the rules for the two lists, rather than merging the actual elements; this option looks particularly attractive when the lists are infinite.

Here's another example: suppose an insurance company has changed computer systems a couple of different times, so that now its records are scattered around on all different kinds of media: floppy discs, magnetic tapes, etc. Instead of writing a program to merge the information stored on these different media into one centralized data base, we could imagine having our data-processing programs call on coroutines to find the data we need to access. There'd be a different coroutine for each medium used.

Yet another example is the notion of data-flow computers, where we regard a computation as a network of nodes, each of which is computed from some nodes that precede it and is an input to some node8 that follow it. We wish to decide how to exploit the possible parallelism to the greatest extent possible, so that our computation is fast and efficient. [References: Paul Kosinski, A data flow programming language, Rep, RC 4264, IBM T. J. Watson Research Center, Yorktown Heights, N. Y., March 1973. Jack Dennis, First version of a data flow procedure language, Tech. Mem No, 61, Lab. for Comptr. Sci, M.I.T., Cambridge, Mass., May 1973. Also, papers by E. Ashcroft at Warwick (about LUCID), by Gilles Kahn at IRIA, and about "Actors" by Carl Hewitt.]

We moved on to consider some interesting facts about the continued-fraction representation of numbers; roughly speaking, the number of digits you know in the decimal part of a real number is the same as the number of partial quotients you know for its continued-fraction representation, Continued fractions also enjoy two nice properties over decimal representations: they terminate if and only if the number they represent is rational; they're periodic if and only if the number they represent is a quadratic irrationality $\frac{(a\pm\sqrt{b})}{c}$, where a, b, and c are integers.

Anent the details of the continued-fraction representation that motivated this problem, DEK remarked that identities like

$$x_0 + /x_1, \ldots, x_{i-1}, 0, x_{i+1}, \ldots / = x_0 + /x_1, \ldots, x_{i-1} + x_{i+1}, \ldots /$$

can be used to find the size of the x's, and that Bill Gosper also works with his own special continued fractions based on powers of two for efficiency. None of these

details, though, should affect the language we're designing for this problem.

We spent some time computing with continued fractions, to get a feel for how they work. Here's the beginning of the continued fraction for π:

$$3+\cfrac{1}{7+\cfrac{1}{15+\cfrac{1}{1+\cfrac{1}{292+\cfrac{1}{1+\cdots}}}}}$$

To compute with this, we form the following table:

i	-1	0	1	2	3	4	5
a_i			3	7	15	1	292
Pi	0	1	3	22	333	355	103993
q_i	1	0	1	7	106	113	33102
c_i	0	00	$\frac{3}{1}$	$\frac{22}{7}$	$\frac{333}{106}$	$\frac{355}{113}$	$\frac{103993}{33102}$

Here's the rule for writing this table: write the partial quotients $a_1, a_2, \ldots$, then for $i \geq 1$ form $p_i = a_i p_{i-1} + p_{i-2}$ and $q_i = a_i q_{i-1} + q_{i-2}$; finally, write $c_i = p_i/q_i$.

Each successive pair c_i, c_{i+1} gives upper and lower bounds on where π lies. We can write the decimal representation for π by using this information: Since $\frac{3}{1} \leq \pi \leq \frac{22}{7}$, we know the first digit of the decimal representation is 3. Then we can subtract 3 from the inequalities, giving $0 \leq \pi - 3 \leq \frac{1}{7}$; this tells us the second digit is 1. When we've exhausted the information given us by one set of bounds, we move on to the next set,

Suppose we want to add π and e. We can illustrate the flavor of the continued-fraction approach by what we do with the decimal representation. We have $\pi = 3.14159\ldots$ and $e = 2.71828\ldots$ Adding the first digits, we know we have $\pi + e = 5 + x$, where $0 \leq x \leq 2$; after adding the second digits, though, we know that $X = 0.8 + y$, where $0 \leq y \leq 0.2$. This tells us that the first digit is 5. Adding the third digits tells us the second digit is 8, since $1 + 4 = 5$.

The only time we have to withhold judgment is when the sum of two digits is 9; then, it's possible that a later addition could cause 1 to carry left, so we need to find out more information before giving the next digit of the number. This problem is, however, quite serious in some cases. For example, consider trying to add ϕ^{-1} and ϕ^{-2}. After getting fifteen digits, we'll know that $\phi^{-1} + \phi^{-2} = .999999999999999$; moreover, there's no prospect that we'll ever get a sum that we'll be forced to carry through all the 9's.

So, we have to make a decision. Since the likelihood of the computer's dropping a bit is probably $> 10^{-15}$, we can say that the probability of the sum $\phi^{-1} + \phi^{-2}$ not starting with 1 is quite negligible, so we might decide that as a general rule, we'll guess that after fifteen successive Q's, we have a 1.

The next stage in DEK's scenario was for someone to activate our routine again, so that it computes the next digit, and finds it to be a 7: .9999999999999997. DEK mentioned three possibilities for handling this case: (1) go into another line of work; (2) give the user an error message; (3) output a digit of "—3" to repair the error and go on. As DRF pointed out, however, we might never discover the error. That's what adds an interesting dimension to the problem: why insist on total correctness of our programs, when the likelihood of their erring is much smaller than the probability, say, of the sun's not rising tomorrow?

We moved on to discuss approaches to Problem 4 itself. DEK remarked that the best way to design a language is to try to write programs in it and see what features you'd like it to have. He also observed that there's an easy cop-out for writing the root-finder for the cubic, namely including an explicit cube root function in the language, and using the general solution *for* cubic equations. He urged us not to take this easy way out, since the real goal is to design a language, not to solve a cubic.

The cubic root-finder is potentially quite useful and interesting, as polynomial equations can be very unstable. The classic example of this case is due to Wilkinson: The roots of the equation

$$\epsilon t^{19} + \prod_{1 \le i \le 20} (t - i) = 0,$$

where ϵ is very small, can be very far away from $\{1, 2, \ldots, 20\}$; many of the roots are complex.

OJD suggested that there's no evidence a priori that we shouldn't redo the entire root computation each time we want a new digit of precision in the answer. DCH replied that if there were some way we could store the state of a computation so that we could return to it if necessary, we'd have better luck dealing with the "9's problem" discussed above.

With respect to solving a system of n linear equations in n unknowns, DEK pointed out that standard Gaussian elimination techniques require us to be careful when a potential pivot element is zero, But how can we test for that in our system? For example, what method could we use to allow our routines to decide that $2(\phi^{-1} + \phi^{-2}) - \sqrt{2}\sqrt{2} = 0$ in a finite amount of time?

ARR pointed out that we might well decide this using an approach similar to our solution to the Q's problem: after a certain number of digits are computed and

are all zero, we declare that the number is zero. MSK observed that this might not be much of a problem since we expect the numbers we'll be working with aren't single values but rather have an associated error tolerance. In other words (due to DRM), we'll be doing arithmetic on *intervals* of numbers, rather than points on the real line.

OJD asked if zero is a coroutine, BIA, who has worked with Bill Gosper on the development of this system, replied that it is, and its representation uses a special "infinity" marker. OJD wondered if anything is known about the distribution of the partial quotients in a continued fraction. DEK replied that, for a random number x, the probability that the n^{th} digit equals a is

$$\lg\left(\frac{(a+1)^2}{(a+1)^2-1}\right).$$

Finally, DEK mentioned the pretty continued fraction representation for e:

$$e = 2 + /1\ 2\ ,1\ ,1\ ,4\ ,1\ ,1\ ,6, 1, 1, 8, 1, 1, 10, \ldots /.$$

Notes from November 9

DEK started class asking what question loomed largest in people's minds when they mull over Problem 4. MSK was the first to volunteer his question. He wondered just how we'd handle the case where the program is asked a question and must reply simply "I don't know." An example of such a question occurs when the program is asked to find the roots of the equation $x^2 + c = 0$ when we only know c to bounds $a < 0 < c < b$. These same bounds on c give us trouble solving $cx = 1$ for x, too.

The general problem of which these are instances is: "When doing interval arithmetic, what do I do if asked to divide by an interval that contains zero?" There are two schools of thought about the appropriate reply. One says we should reply that $\frac{1}{(-1,1)} = (-\infty, -1) \cup (1, \infty)$; the other says we should let the quotient be undefined, so as to be completely rigorous and exclude the possibility of dividing by zero. DEK himself prefers the latter answer, although he admitted that we could debate the point for weeks. DRM asked if we might ever get a result out of computations done by the first rule that consisted of more than two disjoint intervals; DEK assured us that wouldn't happen. NCR pointed out that a significant problem is that the "hole" $[-1, 1]$ could fill up in subsequent operations, like adding $(-2, 2)$.

SOY claimed that it would rarely occur that the program would be need to produce the reply "I don't know," and suggested that we build into the language a facility for handling this case, Specifically, he suggested that we investigate the utility of three-valued logic to the user. In three-valued logic, variables may have

one of three values: T, F, and ?. Here are truth tables for three-valued logic:

$\wedge$	T	F	?
T	T	F	?
F	F	F	F
?	?	F	?

$\vee$	T	F	?
T	T	T	T
F	T	F	?
?	T	?	?

$\neg$	
T	F
F	T
?	?

With these definitions, DeMorgan's law still holds: $\neg(x \wedge y) = \neg x \vee \neg y$. Furthermore, if we assign the numeric values $T = 1$, $F = 0$ and $? = \frac{1}{2}$, "$\wedge$" corresponds to the minimum operator, "$\vee$" to the maximum operator, and "$\neg$" to subtraction from one.

Now, what should our syntax for conditional statements be? CPP observed that we might need three clauses (one for each outcome of the test), so we could use a syntax like the FORTRAN arithmetic IF-statement. JJF suggested the more ALGOL-flavored "**if ... then , , , else .. , perhaps** . . . ," the last branch to be taken if the result of the test was ?. All these suggestions address the problem of expressing the flow of control *of* the language neatly and perspicuously. DEK asked just how much difference people thought such "syntactic sugar" makes, ARR immediately pointed out that it makes quite a difference: his example of SAIL's terminating comments with semicolons, making the likelihood that part of a program is swallowed up as comment much greater, was dear to the heart of almost everyone in the room.

DRM opined that devising the ultimate programming language may be a hopeless task, since people think differently. This prompted JJF to champion PASCAL as a language whose designers distilled a few powerful constructs out of many people's thought processes, giving it a sparse syntax which helps one avoid spending all his time looking up language details in a manual. DEK agreed that the closer programming languages come to being natural expressions of our thought processes, the better they are, but asked what makes programming languages grow to the sizes they do.

CHT made an analogy with natural language: people's need to express ideas fosters the growth *of* facilities in natural language for expressing important distinctions. Eskimos, for example, have over fifty different words to describe snow. He likened efforts to pare programming languages to the barest essentials to ***Newspeak*** in George Orwell's ***1984.*** This led DRM to suggest that a programming language that's an early success is doomed to fail because people will use it and want to extend it to handle the kinds of problems that interest them. OJD agreed and added that there aren't very many people who can expand a programming language sensibly.

All this discussion reminded DEK of his own experience when he designed TEX, and also of the time when PL/I was developed. He said that there are around twenty mutually incompatible desiderata for programming languages, which explains why designing one is so hard: we must trade different goals off against one another. One

of the most important desiderata is a good acronym; he encouraged us all to think of one,

MA1 asked if there were any examples besides the area of programming languages where notation or terminology proliferates, DEK cited factorials as an example of an idea whose notation didn't settle down for a while after the idea was introduced. Binomial coefficients and decimal numbers are still subject to several different notations. And graph theory is an area where there's just a vast overabundance of terminology. [See Fundamental *Algorithms*, vol. 1 of *The Art of Computer* Programming, p. 362.] This is still a problem for those who translate texts into foreign languages: they want to choose felicitous names for the concepts being defined.

DRM suggested that we could avoid choosing words in our language that have particular sentiment for us by using an easily pronounceable concatenation of letters. But DRF suggested that we need to be sparing in naming new concepts; otherwise, we develop a system for expressing our thoughts that's simply unmanageable.

We turned from psychological theorizing to defining the syntax and semantics of the "if . . . then . . . else . . . perhaps . . ." construct. DAS observed that it's quite likely we'd never be able to state for certain that two numbers in continued-fraction representation are equal: we may get a "probably" result, or we may reach the limit of the precision we have in the numbers. JEB observed that the dilemma is similar to the problem of comparing floating-point numbers for equality: we need to establish some tolerance for how close the numbers have to be before we're willing to accept them as "equal." This facility is available, for example, in APL, where it's called SETFUZZ.

MSK suggested that we could use $\|x - y\| \leq \epsilon \|x\|$ as the definition of equality in the language we design, OJD pointed out, however, that equality so defined is not an equivalence relation: transitivity is not guaranteed. DEK noted that it would be nice if hardware designers would put this test for floating-point equality into the hardware. DRF noted that we could still use a macro facility to make the language look nicer and relieve the programmer's need to worry about such nitty-gritty details.

SOY proposed that we create a language feature to set a default limit on the number of partial quotients we'll compute with, BIA, however, pointed out that sometimes we need many digits from one continued-fraction coroutine and only one or two from the others; so, we need more flexibility in our language.

BIA also noted that we'd also have trouble in the proposed system representing a number like Avogadro's number ($\approx 6{,}023 \times 10^{23}$), since all the significance would be in the integer part of the continued fraction. MH pointed out that we could always allow a continued-fraction mantissa with an integer exponent. But MES observed that the language we're designing may simply be unsuitable for computing with this kind of number: we needn't try to make it be "all things to all people."

DCH suggested that we might want to be able to use "normal" real and integer

numbers for speed and efficiency. JEB pointed out that it's reasonable to suggest such an idea, since we're given a routine to translate from a real number with tolerance into a continued fraction. Such "normal" numbers might also be useful as loop indices: we certainly don't need the precision afforded us by continued fractions when we're counting by one! MES asked what conventions would govern conversion among data types; DEK expressed his own preference for the PASCAL convention of strongly typed data types.

Notes *from* **November 14**

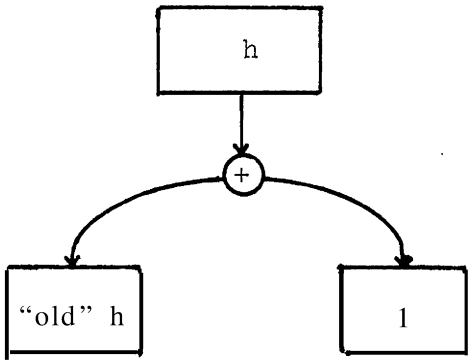
DEK opened class with a letter that recently appeared in a British computer trade magazine urging that data flow computers **really** don't need new languages. This points up the need to design our language **so** that it's a useful extension to existing programming languages, rather than just for the sake of innovation. It's also a shame that there's no system on which we can try programming in the languages we design, since one learns a lot about a language by trying to debug programs written in it. DEK hopes that someone will want to implement a version of the language *we're* discussing. BIA has some routines and notes about such a project, for those interested.

MSK volunteered to show us his proposal for a language. He includes four types of variables: INTEGER, BOOLEAN, COMPARISON (for three-valued logic; CHC suggested the name NON-ARISTOTELIAN), and GOSPER, and the usual operators for comparison and logic. Part of his routine for the cubic root finder might look like this:

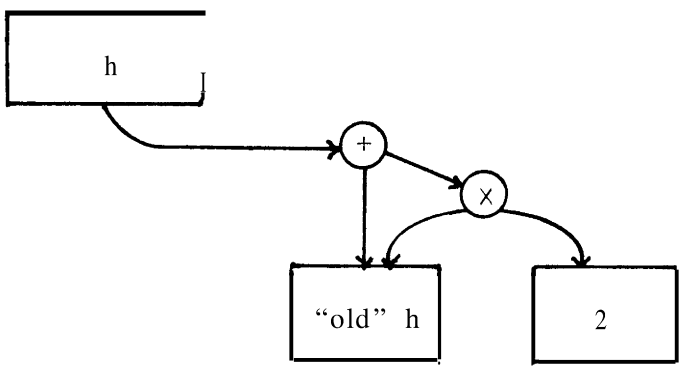
gosper *h*;
h ← 1;
while (f(h) $>_g$ *0)* $\neq$ *T* **do**
    h ← *h* × 2;

(The subscript on the greater-than sign indicates a comparison on Gosper-numbers.) The while-condition looks a little strange, but it's written that way **so** that *h* is doubled until we're sure that *f(h)* > 0. NCR suggested that we might be able to make the syntax somewhat nicer by creating a subtype of COMPARISON.

DAS raised an issue that occupied our discussion *for* most *of* the class, Does it make sense to 6ay "*h* ← *h* + 1" or "*h* ← 2 × *h*" when *h* is a Gosper-number? MSK believes that it does, and interprets it this way: We have *h,* a coroutine, somewhere. To add 1 to it, we instantiate a coroutine for 1 and one for +, and point *h* at the + coroutine, which in turn points to its operands.



MH and DRF asked how this interpretation fits in with the "stepwise" nature of the coroutines: since they're designed to return one digit at a time, won't they need somehow to keep track of who's asked them for digits, so that they can tell each requestor the digit it needs to know next? Another question is raised by **DEK's example** $h \leftarrow 2 \times h + h$:



As JEB phrased it, who turns off the coroutine for the "old h"? Or, are we going to maintain different instantiations of the same coroutine each place we need it?

DMO suggested that each coroutine be required to keep around a list of all the partial quotients it's computed so far, so that it can respond sensibly to requests for more digits. His suggestion was motivated in part by the rather different view his team @MO, RTB, and AMY) takes of the requirements of the language; this is discussed below.

JJF observed that the question of when to get rid of a coroutine, either because it's computed **all** the digits it can or because its value is no longer needed, is very similar to the question of when a variable becomes "dead" in a program. A simple example is the program fragment

$t \leftarrow x;$
$x \leftarrow y;$
$y \leftarrow t;$

which switches the values of x and *y*. If a compiler can tell that the value of *t* is never used again after those statements are executed, it can use a register for *t* during those three instructions' execution, then forget about it,

Then, JJF asked if we're willing to set forth a language that *requires* its compiler to perform such optimizations. It's not clear, though, that our language would require them; also, **ARR** observed that the kinds of variable de-allocation we'd need to do are pretty simple to envision.

NCR pointed out that such considerations suggest that the language we're designing will be compilative, whereas he'd thought it a perfect candidate for interpretation: because the user's knowledge is so important to the "reasonable" running of the program, an interpretive language would offer greater flexibility. DRM asked **if** we plan to take advantage the cyclic structure of the continued-fraction representation for quadratic irrationalities: have a coroutine point to itself once it's reached the cyclic part of the expansion. Such coroutines could be detected by a garbage-collection scheme.

MES expressed his opinion that we shouldn't be talking about "variables" and "assignment" in this language; we need different names because the language doesn't require any computation until output is requested. ML observed that what we've been calling "assignment" simply uses the notion of procedure variables, hence is not that different from familiar ideas. DEK remarked that the notion of assignment is one big difference betwen traditional mathematics and computer science, Von Neumann and Goldstine didn't have the idea of assignments in their original flow diagrams. Even when the language **SIMULA** was first designed, different notations were used for setting pointers and assigning numeric values to variables: the underlying notion of "assignment" being a primitive operation had not been fully assimilated.

WDG mentioned that users of the language might well want to be able to set the accuracy to which they desired the computation to proceed. We might put such options in the input/output facilities of the language. CPP then pointed out that accuracy is not just an issue when output is requested, but whenever the result of a comparison between the values of two Gosper numbers is done. CRT had handled this with explicit language features like "print x **to** *y* places."

DRF observed that the idea of letting users set the accuracy of computation is akin to the way we handle problems of overflow or underflow in floating-point computation: if such anerror occurs, recompile with the offending variables declared to be double precision. (Presumably the program really needs such accuracy, and the range errors are not due to program errors.) He also questioned the feature that the coroutines return single numbers as partial quotients, rather than a range $(x - 61, x + \delta_2)$. DEK hoped, however, that the language would be insensitive to such *lower-level* implementation details.

Responding to **CHC's** question about the point of a coroutine's telling fewer of

the digits in a Gosper number than it knows about, JJF pinpointed the problem as one of how to permit different parts of a computation to use the same coroutine. For example, if $f(h) = ah^3 + bh^2 + ch + d$, then computing the difference $f(h) - f(h+1)$ calls on the h coroutine at least six times,

MH offered the computation of π by series approximation as one in which there are two different "directions" in which we might want more precision:

$$\pi = 1 - \frac{1}{3} + \frac{1}{5} - \frac{1}{7} + \cdots.$$

We could ask for more terms in the expansion, or for greater precision in those we already have.

JEB noted that most of the discussion had centered around implementation details, rather than the user's perspective. Should a user be expected to understand the issues we've discussed in order to be able to use the language? Of course, taking this argument to extremes, we could simply say our language will be FORTRAN or PASCAL with real variables implemented as Gosper numbers.

RTB explained that his team sees the computation as one driven by comparisons and output requests, hence would tailor their language to a user who knows the type of variables with which he wants to compute, but who doesn't want to worry about all the details of setting up such a computation. In this vein, WOL proposed that if there's not enough accuracy at some point in the computation to decide the result of a comparison, we let the user know what we know and allow him to change his program or direct the computation some other way.

This idea suggests that one of the most important features of the language will be a nice facility for letting the user know what his program is doing, especially at run time. DAS offered this illustration of how the computation proceeds: we have a big pot of spaghetti, and when we yank on one noodle (for output), many of the others may get stirred up,

Notes from November 16

DCH and ARR were the first to volunteer to show their proposed language; they had extended the language PASCAL to handle Gosper numbers. Here's their program to find one root of a cubic equation:

```
program findroot;
  var
    AU, Al, AZ, A3, (* coefficients for the function *)
    X, NEWX, (* approximations to the root *)
    FofX, FprimeofX : real; (* approximation to F(x), F'(x) *)
    Count, Maxcycling : integer; (* cycle counter and stop *)
  begin
```

```
Setaccuracy (10);
Read (A0, Al, A2, A3); (* read in coefficients *)
Read (NEWX); (* read first approximation *)
Read (Maxcycling); (* read maximum number of cycles *)
Count := 0;
loop
    Count := count +1;
    if Count > Maxcycling then exit; (* stop if too long *)
    X := NEWX;
    FofX := A0 + X ×( A1 + X ×( A2 + X × A3 ));
    FprimeofX := A1 + X ×(2 × A2 + X ×3 × A3);
    NEWX := X — FofX / FprimeofX;
    if Abs(NEWX—X) = 0
        then exit (* stop if accurate *)
        else
        indoubt exit (I they are close, anyway *)
    end;
if Count > Maxcycling
    then writeln ('Not enough cycles. Root between',X,'and',NEWX)
    else writeln ('Root at',NEWX:20:10)
end.
```

The additions to PASCAL are the three-way conditional statement "if . . . then . . . else . . . **indoubt** , , .," and the capacity to set accuracy dynamically. They also avoid the weird syntax problems on WHILE conditions by means of the loop . . . exit construct. Finally, there's a procedure to release more digits from a particular coroutine than are required by the general Setaccuracy statement.

CPP pointed out the first problem with this method, namely that it incorporated Newton's method, which simply may not work if the initial approximation to the root is not good enough, But that's not really a problem with the proposed language (the two problems were given for wncreteness' sake), so we punted that issue.

IAZ asked if there's a mechanism for undoing statements, for example if the result of a comparison is later falsified by more digits coming out of a wroutine. This language includes no built-in mechanism, but expects the user to enter a debugger to deal with such problems.

DEK wondered if **Setaccuracy** obeys the block structure of PASCAL. Such a LIFO structure would be much nicer than globally resetting the parameter, as SAIL's SETFORMAT command does. DEK also liked the **indoubt** clause. He summarized the underlying philosophy of this approach as a conservative extension to PASCAL.

NCR proposed a similar language structure, but handled accuracy **considera**tions by attaching an explicit accuracy argument to each Gosper number when it's

used in computation, conditional expressions, or output statements, Thus, binary operators become Gosper functions with more than two arguments. Then, if we come to a point in the computation where we want greater accuracy, we redo a lot of computations to get greater accuracy where required. This viewpoint fits in with the data flow tree we've seen before.

NCR also mentioned what happens when we change the last known partial quotient by one:

$$/x_1, 1$$ [illegible]

is always the reciprocal of an integer. DEK mentioned that the denominator is related to the continuant polynomial of the first k partial quotients. For example,

$$q_4(x_1, x_2, x_3, x_4) = x_1x_2x_3x_4 + x_3x_4 + x_1x_4 + x_1x_2 + 1,$$

the number of terms being a Fibonacci number. This theory is related to finding the best rational approximation to a real number using small denominators,

DRF suggested that instead of explicitly setting accuracy we just run the program with increasing accuracy until it finds the answer desired. In other words, embed the above program in a huge while-loop that increases the accuracy to which the computation is performed until we have the answer.

BIA mentioned that the 9's problem hasn't really been addressed by these examples. But **JEB** pointed out that Gosper's method guesses for us, so the problem is not one for us to solve anyhow. We discussed methods for notating NCR's **accuracy**-setting on operators: DEK mentioned that no one has a really good **ternary** operator notation. However, the average number of operations in a statement is around one, so perhaps we shouldn't worry about this problem too much either,

IAZ proposed a radically different kind of language. He would write programs as data-flow graphs, and at every comparison, try to do the computation required by ***both*** possibilities for the value of the Boolean expression. This method could handle arrays by lots **of** copying. **IAZ** hoped he could do without an **indoubt** clause in **his** conditional statements, but DEK pointed out that inaccuracy in input may necessitate such a feature,

Notes on Solutions to Problem 4

Everyone decided to extend either PASCAL or **SAIL** to handle Gosper numbers. Of course, people differed on details, and several interesting issues *were* addressed by only a few of you.

Everyone added **a** type GOSPER, and most kept INTEGER as a separate type; only a few kept REAL as a distinct type, however, Although the problem statement called for it, only half the class made explicit provision for the user to specify the accuracy of his input data. Half the class included a facility for the user to write his

own coroutines to specify variables. (This would be useful for e or $\sqrt{2}$, for example.) Most teams provided user-settable accuracy, either globally (SETACCURACY 10 DIGITS) or in each statement (PRINT x TO 7 DIGITS); some allowed both.

One issue that sparked a lot of discussion was what to do when a test for equality is indeterminate. **JEB** and **JJF/MH** decided to call the result "true." **MAI/DRF** and ML increased flexibility by adding indeterminacy as an operator: for example, "if 2.999 **<?3"** means "if 2.999 < 3 or you can't tell." NCR and **CPP/CHT/WOL** wanted to run their languages in an interpretive environment, so that the user could direct the computation himself if an indeterminate condition arises. Everyone else extended the if-statement to a three-way branch.

The dominant view of programs written in the language was as computation networks driven by comparisons and output requests. A couple of you (AFB, NCR, DAS, **MES)** considered with some care just how big this network can grow when loops are involved, AFB put it best when she observed that it "boggles the mind."

Very few people discussed the difficulties with backing up to recover a computation should one discover that a Gosper coroutine had guessed wrong. Since that's one of the harder issues confronting a user, it's surprising that so many omitted it from their discussion.

Only four teams suggested names for their languages: MES offered GAL or **GospeL** for Gosper Arithmetic Language. **CPP/CHT/WOL** proposed CORDIAL for **CORoutine DIrected** Accuracy Language. **RTB/DMO/AMY** suggested YAWL for Yet Another **Worth**$_{\text{less}}^{\text{while}}$ Language. And **WDG/DRR** overheard the user discussing his computing bill and called theirs **@/?**&%$'.**

Problem 5. Kriegspiel **endgame** (due December 5)

In this problem we are concerned with the game of chess where there is a white king and white rook against a black king, An easy checkmate, you say; but there's a twist: The white player does not know where the black king is.

The game starts with the white king and king rook at their normal opening positions, while the black king may be anywhere else (but not in check); White moves first. Whenever White makes a move, he or she receives one of the following replies:

a) Checkmate, you win.
b) Stalemate, it's a draw.
c) That move of yours is illegal because it puts your king next to mine; try again.
d) Your move put me in check, but it was not checkmate **so** I have made my next move.
e) Your move did not put me in check, and I have made my next move.
f) I took your rook. It's a draw.

It is possible for White to force checkmate in finitely many moves, but the winning strategy is not easy to discover, so a good Black will usually be able to escape.

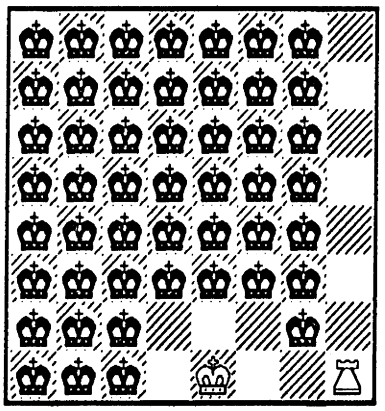
Your problem is to design and program a Black strategy that plays "intelligently"; in particular, your program should be able to survive at least 100 moves against two different White programs supplied by the grader. (These White programs will, of course, be incorrect attacking strategies, typical of what a player might try if he doesn't know the secret of winning.) You will not be able to see the grader's second program until the final contest actually takes place (nor will he look at your program); but the grader's first program will be available to aid you in preliminary testing.

This game is different from true Kriegspiel in that Black does know the positions of White's pieces. Furthermore, Black does not have to be committed to particular king moves at each step, he or she is allowed to imagine being in any set of possible positions consistent with what White has been told. After seeing what White does, Black has the right to decide what the past history actually was,

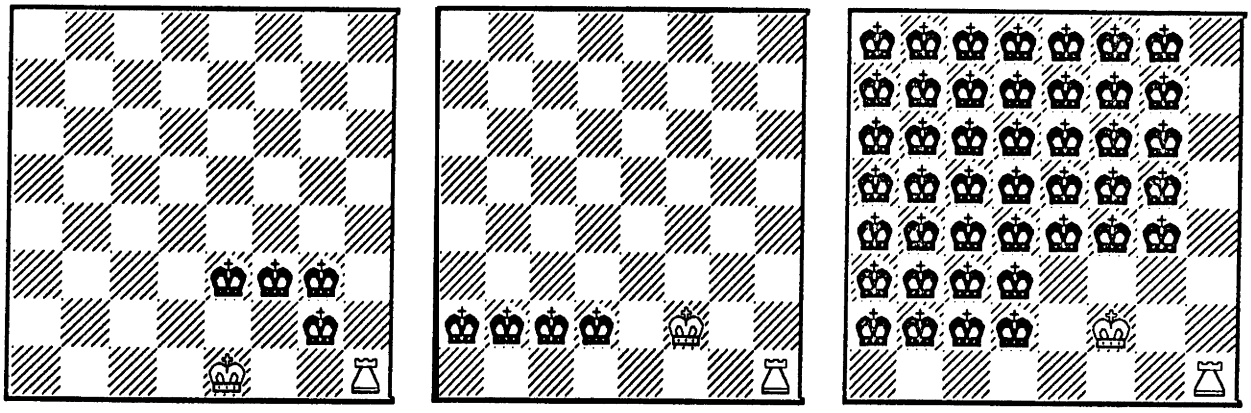
Notes from November 21

DEK started the class off asking why he'd posed Problem 5 at all. JEB said he assumed that it was the Artificial Intelligence problem of the quarter, but DEK had hoped to provide problems less closely identified with one area than that, MES suggested that one of the substantial issues to be faced in this problem is that of representing the knowledge one has about the board situation and the heuristics he or she plans to use in play. DRM pointed up the related problem of devising a good data structure for knowledge. DEK partitioned the possible strategies into two kinds: providing **a** huge game tree of moves and responses or deciding on a move based on the current board state and some look-ahead. Also, this problem should be fun to work on.

Since **JEB** claimed to have discovered a winning strategy for White, he played against the **class;** the rest of us took turns replying to his moves, The initial board position (showing all possible Black King positions) is:



After **JEB's** first move **"k-f2,"** we may respond with one of "illegal," "check," and "okay," leading respectively to



Here is the result of the first game:

1.	k-f2		k-e5	19.	k-h3	26.	**k-e2**
	r-h5		k-d3	20.	k-g2	27.	k-d2
2.	r-a5	9.	k-e3		k-h2		k-d3
3.	k-d2	10.	r-d5	21.	k-g2		**k-e1**
4.	k-c3	**11.**	k-f3		r-h3	28.	k-d2
5.	k-d3	12.	**r-e5+**	22.	k-g2		r-h2
6.	**r-c5**	13.	**r-e3**	**23.**	k-f2	29.	**k-d1**
7.	k-e3	14.	k-f4		k-f3	30.	k-c2
	k-e4	15.	r-13	**24.**	k-f2	31.	r-h3
	k-d4	16.	k-g4	**25.**	k-e2	32.	**r-a3#**
8.	k-e3	17.	r-h3		k-e3		
	k-e4	18.	**r-g3**		**k-f1**		

MES wanted to resign at move 20, fearing the worst, but was dissuaded by DEK, who hoped **JEB** might trip up at the end game and stalemate Black.

The second **game started** differently, because almost nobody thought that **DEK's** first move (to declare **"k-f2"** illegal) was very smart:

1.	k-f2	8.	r-e4	**15.**	k-h4	22.	**r-f3**
2.	k-f3	9.	k-e3	16.	k-h5	23.	k-d6
3.	**r-g1**	**10.**	k-13	17.	r-h4	24.	**r-e4+**
4.	k-e3	11.	r-g4	18.	**k-g6**	25.	**r-e6**
5.	**r-f1**	12.	k-g3	19.	k-f6		
6.	k-d3	13.	**r-h4**	**20.**	r-g4		
7.	**r-e1+**	14.	r-g4	**21.**	**k-e6**		

At this point, a tremendous sense of **déjà** vu overwhelmed us, so we recalled our response to move 21, changed it to "illegal," and continued:

21. **k-e6**	k-e5	**24. r-f4+**	**27.** k-g5
k-f5	**k-d4**	**25.** r-f6	**28, r-h6**
22. k-e6	**23. k-d5**	**26. k-f5**	

In response to **DMO's** inquiry asking what Black programs should do given that White has a guaranteed winning strategy, DEK assured us that the grader's White program (against which the class programs will be tested) will not incorporate a perfect strategy.

DEK then began to describe an approach to writing a Black program that would last a long time against a White program, The method is called alpha-beta pruning, and is described in **D.** E. Knuth and R. W. Moore, **Artificial *Intelligence* 6 (1975),** 293-326. It assumes that the opponent will be using a good strategy also. To play a game **using** alpha-beta pruning, we need a function that evaluates board positions, f(P). We then define the function

$$F(P) = \begin{cases} f(P), & \text{if } P \text{ is a leaf in the game tree;} \\ \max\{-F(P_i) \mid P_i \text{ is a descendant of } P\}, & \text{otherwise.} \end{cases}$$

Here's a procedure that calculates *F(P)* from the above definition:

```
integer procedure F( board position P);
  begin
  -integer m,i,t,d; [d is number of descendants of P]
  if d = O
     then F t f(P)
     else
        begin
        m t - 0 0 ;
        for i ← 1 thru d do
           begin
           t ← —F(Pi);
           if t > m then m ← t
           end;
        F t m
        end
  end
```

We can do better than this by defining an auxiliary function whose value is indeterminate **in certain cases:**

$$F1(P,\beta) = \begin{cases} F(P) & \text{if F(P)} < \beta \\ \geq \beta & \text{if } F(P) \geq \beta \end{cases}$$

Then we have $F(P) = F1(P,\infty)$ and we can use this procedure:

```
integer procedure F1( board position P, β);
    begin
    integer m,i,t,d; [d is number of descendants of P]
    if d = 0
        then F1 ← f(P)
        else
            begin
            m ← −∞;
            for i ← 1 thru d do
                begin
                t ← −F1(Pi, m);
                if t > m then m ← t;
                if m ≥ β then done
                end;
            F1 ← m
            end
    end
```

The full alpha-beta method uses a more complicated auxiliary function:

$$F2(P, a, \beta) = \begin{cases} \leq \alpha & \text{if } F(P) \leq a \\ F(P) & \text{if } \alpha < F(P) < \beta \\ \geq \beta & \text{if } F(P) \geq \beta \end{cases}$$

Analogously, we have $F(P) = F2(P, -\infty, \infty)$ and we can use this procedure:

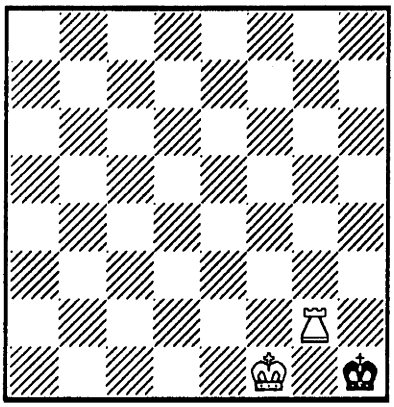
```
integer procedure F2( board position P, a, β);
    begin
    integer m,i,t,d; [d is number of descendants of P]
    if d = O
        then F2 ← f(P)
        else
            begin
            m t a
            for i ← 1 thru d do
                begin
                t ← −F2(Pi, −β, −m);
                if t > m then m ← t;
                if m ≥ β then done
                end;
            F2 ← m
            end
```

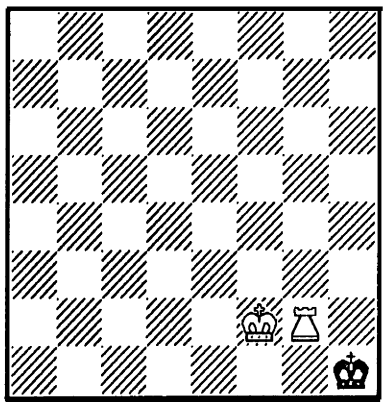
end

This procedure, while it requires less computation than the other two, takes at least four levels in the game tree for the savings to be observed,

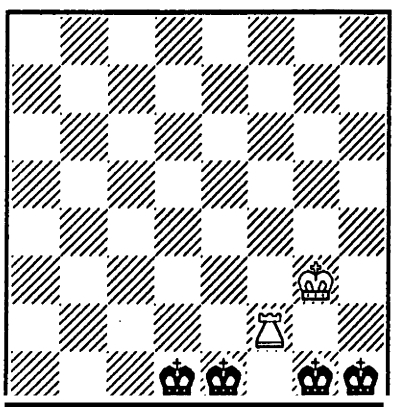
Notes *from* **November** 28

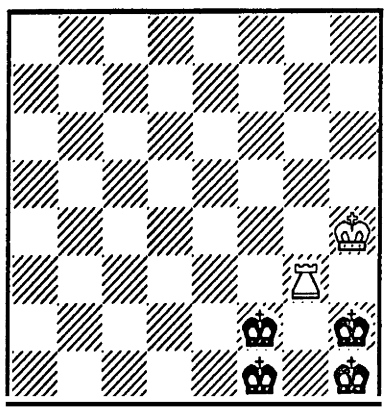
DEK asked for examples of stalemate positions. WDG provided these two:





Here are two positions from which it's possible for White to stalemate, although the Black King is not known to be confined to one position:





On the left, if White moves **r-g2,** he may stalemate the Black King at hl. On the right, White's move k-h3 may stalemate the Black King at hl.

JEB mentioned that he can guarantee a White mate in fewer than 40 moves. This means that our Kriegspiel problem can be completed within chess's regular fifty-move rule,

SOY asked how to represent the chess board and move information in a computer program. DEK replied that our task is easy compared to regular chess: there are no esoterica like en ***passant*** captures or castling to **worry** about. He also mentioned that the difference between good and excellent programmers is often the representation

they choose to solve problems.

CHC agreed with this latter point and emphasized that a poor choice of data structure may obscure the path to solution of a problem, DEK cited the first problem of the quarter as an instance of such an effect. He mentioned that it was well within the realm of possibility to solve the problem with no explicit representation of the chess board in memory at all.

At NCR's prompting, we discussed how hard it would be to develop a representation scheme that allowed Black to decide its "actual" board position and the history leading thereto, JEB suggested we use a list of the possible moves to get to each possible position of a Black King. DEK remarked that we could store each possible board position as a reply is made to White moves. DMO asked how hard it would be to move "backwards" through the history of Black responses. It is apparently just as hard as moving "forwards," and neither is easy.

This discussion is relevant not only because the Black programs need not be committed to a move at each response, but because it bears on the depth of exploration in alpha-beta pruning. If we can decide easily how to switch from one board position to the next, we can get away with keeping just as many board positions as the depth of the alpha-beta tree we're searching. This could be very important because, as IAZ pointed out, there may be twenty-two possible White moves at almost every step.

This latter point prompted CPP to observe that most White moves at a step aren't "reasonable": they lead to instant Rook capture, or just increase the freedom of the Black King. The alpha-beta method takes care of some of this, though, pruning out branches of absurd moves,

DRM asked if an expert player could spot opponents' weaknesses, and exploit them in its play; it might be worthwhile to try to build such a "psychological" facility into a Black program. DEK remarked that good computer chess programs can do this, but only in a limited way: they understand piece advantage much better than positional advantage.

This discussion gave rise to a short history of computer chess. DEK mentioned that David Levy had lost one game to the computer and drawn another in the match he played in August (which he ultimately won). But the program against which he played didn't incorporate any especially elegant insights into chess: it was faster because it ran on faster hardware, so it could search more levels of the game tree.

DEK remarked that game-playing programs are generally very resilient to errors: as long as a program plays legally, one may be led to believe simply that he's incorporated an insufficiently sophisticated strategy, rather than that he's implemented a sophisticated one incorrectly. He also mentioned that mistakes made in research can be a fruitful ground for insight into solving problems: we learn a lot

by solving the simpler problems that arise when we overlook some aspect of a problem.

Notes **from** November 30

We spent some time discussing the representation method for the board of possible Black King locations. DAS had mentioned at the end of last class that he could do it with eight words, each of which contained one rank of the board. If the old **board is in** $A_1, \ldots, A_8$, we can create the board on which every king in the A's moves all possible ways as follows: First, define

$$A_i^* = (A_i \uparrow 1) \vee (A_i \downarrow 1).$$

Then, here's the i^{th} rank *of* the new board:

$$B_i = A_{i-1}^* \vee A_{i+1}^* \vee A_i^* \vee A_{i-1} \vee A_{i+1}.$$

Now, of course, we must ensure that the Black Kings aren't in check or an illegal position relative to the White King and Rook. DAS suggested that we could take **another** eight words and zero out the bits corresponding to squares next to the White King or on the same **rank** or file as the White **Rook. MES** mentioned that we could achieve this effect with less storage by keeping track of the White pieces' locations in terms of their coordinates. Ultimately, we followed **ARR's** suggestion to 'use an integer telling the piece's rank and a word with the appropriate bit set to tell the piece's file, Say X is the word and Y the rank-integer for the King, and x and y the corresponding information for the Book. Then we can get rid of Black Kings in the White King's immediate vicinity by the following computation:

$$C_i = \begin{cases} B_i \wedge \neg(X \vee (X \uparrow 1) \vee (X \downarrow 1)) & \text{if } \|i - Y\| \leq 1 \\ B_i & \text{otherwise} \end{cases}$$

To get rid of the Black Kings attacked by the White **Rook,** we tried this first:

$$D_i = \begin{cases} C_i \, A \, x & \text{if } i = Y \\ C_i \wedge \neg x & \text{otherwise} \end{cases}$$

This works only *if* the White King and White Book are neither on the same rank nor the same file ($x \neq$ X and y $\neq Y$).

MSK pointed out the problem with computing when x = X or y = Y: the Rook's attacking power may be truncated by the King's being in the way. This means a little more work to write down the Rook-elimination equations in these cases. If X = x, we define $D_i = C_i$ when $i \leq$ Y $< y$ or $i \geq$ Y $> y$, otherwise use the same equations as above. Similar changes occur if Y = y.

All this bit fiddling prompted NCR to ask whether this wouldn't be easier to compute on an array machine, DEK granted the point that such machine8 are very

powerful-for example, they enable us to multiply $n \times n$ bit matrices together in O(log n) steps, using $O(n^3)$ bits per step. NCR added that with fast enough big memories, many artificial intelligence techniques are surprisingly powerful.

MSK presented his idea for a static board evaluation function: Number each square between 1 and 10, depending on its distance from the center of the board, higher numbers being toward the center. Then, in deciding **what response to give the** White player, try to maximize the value of the function on the board. If two are tied for maximum, take the one that maximizes the smallest rectangle containing all **the** Black Kings and one **corner** of the board. One way to compute this using Boolean operations would be to use ten **64-bit** words, A_i containing a one for squares of value i on which a Black King might appear.

JJF pointed out that this algorithm might miss some board positions in which the possible Black King locations are spread all over the board, but on low-value squares, Such positions are indeed terrible for White to try to mate from, but don't seem to arise much in practice. This objection seems to be a special case of **JEB's** idea that the evaluation function should include the White piece locations in deciding the value of a position, because some moves may make the Black possible location set smaller, but harder for White to confine in one move.

Notes from December 5

Here are the results of the playoff between the grader's White program and the class's Black programs, Some teams also played their **programs** against **JEB's** guaranteed-win White strategy.

While preparing the notes for this session, the grader **noticed** that several of **the** games played were identical. Therefore, **a** game may be recorded with more than one title heading,

We use a modified algebraic notation to record the progress of the games. The move number is not incremented until a move has been declared legal by Black. The symbol "+" by a move means it placed Black in check; "=" means a draw; "#" means mate.

CVW (White) **vs.** CPP, CHT, and WOL (Black)

1.	k-f2	13.	**k-e5**		k-e6	27.	k-d7
2.	**r-g1**		r-g4		**r-f5**		r-16
3.	k-f3	14.	k-e5	20.	k-e6	28.	k-d7
4.	r-g2		r-e4		k-e5		k-d6
5.	k-e3	15.	**k-e6**		**r-e5**		**r-d6**
6.	r-f2		**k-e5**	**21.**	**k-e7**	29.	k-d7
7.	k-e4		**r-f4**		r-e6		r-d7
	r-f4	**16.**	k-e5	22.	*k-f7*	30.	**k-d8**
8.	k-f3		k-e4	**23.**	k-e7	31.	r-c7
9.	k-g4		r-e4		**r-g6**	32.	r-e7
10.	k-g5	17.	**k-e6**	**24.**	k-e7	33.	k-c7
11.	k-f6		**r-e5**		**k-e6**	34.	**r-e6**
12.	**k-e5**	18.	k-f6	25,	k-e7	35.	**r-a6#**
	k-f5	19.	**k-e7**	26,	**r-d6**		

CVW (White) **vs.** DCH and **ARR** (Black)
CVW (White) vs. DRF and MA1 (Black)
CVW (White) vs. MSK and NCR (Black)

These three programs all fight for the middle of the board. The only difference in their behavior came at the end of the game, **DRF/MAI** and **MSK/NCR** took the White Rook at move 42. **DCH/ARR** replied check; after it called White's **"k-c7"** illegal, White moved **"r-c6"** and **DCH/ARR** claimed to have captured the White Rook, which is impossible,

1.	k-f2		k-f5		k-c4	32,	k-b5
2.	**r-g1**	15.	**k-e6**		r-d3	33.	**k-c6**
3.	k-f3		**k-e5**	25.	k-c4		k-b5
4.	r-g2		r-f4		k-d4	34.	k-c5
5.	k-e3	**16.**	k-e5		k-c3	35.	**k-c6**
6.	**r-f2**		k-e4	26.	k-c4		**r-d6**
7.	**k-e4**		**r-e4**	27,	k-c5	36.	k-d5
	r-f4	17.	k-e6		r-d3	37.	k-e6
8.	k-f3	18.	**k-d5**	28,	k-c4	38.	k-e7
9.	k-g4		r-f4	**29.**	k-c5	**39.**	k-d7
10.	k-g5	19,	k-e5		r-d5		r-f6
11.	k-f6	20.	**k-d5**	30.	k-b5	40.	k-d7
12.	k-e55		k-e5		**k-c5**		**k-d6**
	k-f5	21.	k-e4		r-d4		**r-d6**
13.	k-e5	**22.**	k-d3	31,	k-c5	41,	k-d7
	r-g4	23.	k-c3		k-d5	42,	**r-a6=**
14.	k-e5	24.	k-b4		**r-d5**		

CVW (White) **vs. IAZ** (Black)
CVW (White) vs. RTB, DMO, and AMY (Black)

1.	k-f2	10.	r-e3	**19.**	r-a5	28.	k-d6
2.	**r-g1**	11.	**k-d5**	20.	k-c5	29.	k-d5
3.	k-f3	12.	r-e4	**21.**	**k-b6+**	30.	**r-c6**
4.	r-g2	13.	k-c5	**22.**	**r-c5**	31.	k-e5
5.	k-e3	14.	**r-d4+**	**23.**	**r-c7**	**32.**	**r-d6**
6.	r-12	**15.**	**r-d5**	24,	k-b7	33,	k-e4
7.	k-e4	**16.**	k-c4	25,	k-c8	34,	r-d5
8.	**r-f3**	17,	**r-b5**	26.	**k-d8**	35,	k-f4
9.	k-d4	18,	k-b4	27,	k-e7	**36.**	**r-e5+**

We are now in a position symmetric with that after move **14.** The program lasted 100 moves.

CVW (White) **vs. JJF** and MH (Black) (evaluation function)

1.	k-f2		k-f4		k-e5		r-f2
2.	**r-g1+**	9.	**k-f5**	13.	**r-f4+**	*20.*	k-g4
3.	**r-e1**		r-e3	14.	r-d4	**21.**	r-f5
4.	k-13	**10.**	**k-f5**	15,	*k-f5*	*22,*	k-g5
5.	r-e2		r-e5		r-f4		r-f3
6.	k-f4	11.	k-g5	16,	k-e4	*23,*	k-g5
	r-e4		k-f5	**17.**	k-f3		k-f5
7.	k-g4		r-e4	18.	k-g3	*24.*	k-g5
8.	k-f5	**12.**	k-15	**19.**	k-g4		**r-h3#**

CVW (White) vs. **JJF** and MH (Black) (random)

1.	k-f2	*5.*	**r-c3**	**9.**	r-f3	13,	r-e3
	r-h8	**6.**	r-e3	10.	k-f2	14.	k-g3
2.	r-c8	**7.**	k-e2		r-d3	**15.**	**r-e1#**
3.	r-c1		r-c3	11.	k-f2		
4.	**k-d2+**	**8.**	**k-e2**	12,	r-g3		

CVW (White) vs. **JJF** and MH (Black) (random)

This game exposed *one of* the errors in the grader's program: he moved his Rook off the board!

1.	k-f2	*7.*	k-g4	**12.**	k-d7	16,	**r-a6**
2.	**r-g1+**		k-f4	13,	k-e7	**17.**	k-17
3.	**r-e1**	**8.**	k-f5		**r-c6**		**r-f6**
4.	k-f3	**9.**	**k-f6**	**14.**	k-e7	18.	k-f7
5.	r-e2		**r-e6**		**k-e6**	**19.**	**r-i6???**
6.	k-f4	10.	**k-e5**		r-e6		
	r-e4	11.	k-d6	15,	k-e7		

CVW (White) **vs.** MES and DAS (Black) (version 1)

This program also managed to take White's Rook.

1.	k-f2	**9.**	k-d4		k-f3	**20.**	k-g3
2.	**r-g1**	10.	**r-e3+**		r-e2		**r-f1**
3.	k-f3	**11.**	**r-e5+**	17,	k-f3	21,	k-g3
4.	r-g2	12,	**r-e3**		**k-e3**		k-f3
5.	k-e3	13.	k-d3		r-e3		r-f3
6.	**r-f2**	14.	k-e2	18.	k-g3	*22.*	k-g3
7.	k-e4	15,	k-f2		r-13	*23,*	r-15-
8.	**r-f3**	16.	**k-g3**	19,	k-g2		

CVW (White) **vs. MES** and DAS (Black) (version 2)

1.	k-f2	**8.**	k-e2	**20.**	k-g5	26.	k-g5
	r-h8	**9.**	**k-d3**		k-f5		r-g5
2.	r-c8		r-cl		r-e6	27,	k-h5
3.	r-cl	10.	k-d2	21.	k-f5	28,	r-g4
4.	k-d2	11.	k-d3		k-e5	29.	k-h4
	k-dl	12.	r-c2		**r-e5**		r-g6
5.	k-d2	13.	k-d4	22.	k-g5	30.	k-h4
	r-c2	14.	**r-c3+**		r-f5	31.	r-g3
6.	k-e2	15,	r-c5	23.	**k-g6**	32.	r-g5
	k-d2	16.	k-e4	24.	k-g5	33.	k-g3
	r-cl		r-e5		r-f7	34.	**r-f5**
7.	k-d2	17.	k-d5	25.	k-g5	35.	**r-f1#**
	k-c2	18.	k-e6		k-15		
	r-c2	19.	k-16		**r-f5**		

CVW (White) **vs. WDG** and **DRR** (Black)

1.	**k-f2**	**8.**	k-e4	**15.**	k-c4	22,	k-f3
2.	**r-g1**	**9.**	r-d3	**16.**	k-c3	**23.**	k-e2
3.	k-f3	10.	k-e5	17,	k-d2	24,	k-d3
4.	r-g2	**11.**	**r-d4+**	18.	**k-e3**	25,	k-c3
5.	k-e3	12,	**r-f4+**	**19.**	k-f3	26,	**r-d4+**
6.	**r-f2+**	13.	**r-d4**	20.	**r-e4+**		
7.	**r-d2**	14.	k-d5	**21.**	k-f2		

We are now in a position symmetric with that after move **20.** The program lasted 100 moves,

JEB (White) vs. CPP, CHT, and WOL (Black)

1.	**k-f2**		k-c4	23.	k-f6	31.	k-g5
2.	k-f3		k-b6		**r-e4**		k-f5
3.	**r-g1**	15.	k-c5	24.	**k-f5**	32.	k-g5
4.	k-e3	**16.**	k-d5		k-e5		k-g4
5.	**r-f1**		k-d4	25.	k-f5	33.	k-g5
6.	k-d3		k-c6		k-f4		k-h4
7.	**r-e1+**	17.	k-d6	26.	**k-f5**	34.	k-g5
8.	r-e4	18.	r-c4		k-g4	35.	**k-g6**
9.	r-c4	19.	k-d5	27.	k-f5		**k-f6**
10.	k-c3	**20.**	**k-e5**	28.	**k-f6**	36.	**k-g6**
11.	k-b4		k-e4		k-e6	37.	k-f7
12.	k-b5		**k-d6**	29.	**k-f6**		**k-h6**
13.	r-b4	21.	**k-e6**	30.	k-g6	38.	**r-f8#**
14,	k-c5	22.	r-d4		r-f4		

JEB (White) vs. **DRF** and **MAI** (Black)

1.	k-12	13.	**r-b4**	21.	k-d5		**k-d8**
2.	k-f3	14.	k-c5	22.	k-c5	30.	k-e7
3.	**r-g1**	15.	k-d5	23.	k-b6	31.	k-f7
4.	k-e3		k-d4	24.	k-b7		**k-f6**
5.	**r-f1**		k-c6	25.	r-b6		k-e8
6.	k-d3	16.	k-d6	**26.**	k-c7	32.	k-f7
7.	**r-e1+**	17,	r-c4	27.	k-d7	33.	**k-g6**
8.	r-e4	18.	**k-d5**		k-d6		k-18
9.	r-c4	**19.**	**k-e5**		**k-c8**	34,	**r-h6#**
10.	k-c3		k-e4	28.	**k-d7**		
11.	k-b4	20.	**k-e5**	**29.**	k-e7		
12.	**k-b5**		**r-c6+**		k-e6		

JEB (White) vs. **IAZ** (Black)

JEB's program moved its **Rook** off the board in this game.

1.	k-f2	10.	**k-c3**	**19.**	**k-e5**	26.	k-g5
2.	k-f3	11.	k-b4	20.	k-e6	27,	k-g6
3.	**r-g1**	12.	**k-b5**	21,	r-d4		k-h5
4.	k-e3	13.	r-b4	22.	**k-f6**	28,	**k-g6**
5.	**r-f1**	14.	k-c5		r-e4		**r-g4+**
6.	k-d3	15.	k-d5	23.	k-f5	29,	**k-i4???**
7.	**r-e1+**	16.	**k-d6**	24.	**k-f6**		
8.	r-e4	17.	*r-c4*	25.	k-g6		
9.	r-c4	18.	k-d5		r-14		

JEB (White) vs. JJF and MH (Black) (random)

1.	*k-f2*		k-f2	**9.**	r-e3	12.	k-g5
2.	k-f3	**5.**	k-f3	10.	k-f5		**k-f5**
	r-d1		**r-e1**		k-e5	13.	k-g5
3.	k-e2	**6.**	k-f3		k-g4	14.	k-g6
4.	k-e3	**7.**	r-e2	11.	k-f5	15.	k-f7
	k-d3	8.	k-f4		r-f3	16.	**r-h3#**

JEB (White) vs. MES and DAS (Black) (version 1)

1.	k-f2	11.	**k-b4**	17.	k-c6		k-e6
2.	k-f3	12.	k-b5	18.	k-d6		k-d8
3.	**r-g1**	13.	r-b4		**k-d5**	23.	k-e7
4.	k-e3	14,	k-c5		k-c7		r-b7
5.	**r-f1**		k-c4	**19.**	k-d6	24.	k-e8
6.	**k-d3**		k-b6		**r-b6**	25.	k-f7
7.	**r-e1+**	15.	k-c5	20.	k-d7	26.	**r-b6**
8.	r-e4		r-b5		k-d6	27.	**r-h6#**
9.	r-c4	16.	**k-c6**	21.	k-d7		
10.	k-c3		**k-c5**	22.	k-e7		

JEB (White) vs. MES and DAS (Black) (version 2)

1.	k-f2	**6.**	k-d2	11.	k-d3	15.	k-c2
	r-h5		**k-f2**		k-d4		k-c3
2.	r-a5		r-84		k-e2		**k-d1**
3.	k-d2	7.	k-d2	12.	k-d3	16.	k-c2
	r-a4		**k-f2**		r-13		r-f2
4.	r-h4	8.	k-e2	13.	k-d2	**17.**	k-c2
5.	k-12	**9.**	r-f4		k-d3	18.	r-f3
	r-b4	10.	k-e3	14.	k-d2	19.	**r-a3#**

JEB (White) vs. RTB, DMO, and AMY (Black)

1.	*k-12*	8.	k-c3	15.	**k-d5**	22.	**r-c6**
2.	k-f3	**9.**	*r-e2*	16.	k-c5	23,	k-d7
3.	**r-g1**	10.	k-d3	17.	r-e4	24.	**k-e7**
4.	**k-e3**	11.	**k-d4**	18.	**k-c6**	25.	k-f7
5.	**r-f1**	12.	k-c4	**19.**	**r-d4+**		**k-f6**
6.	k-d3	13.	r-e3	20.	**r-d6**	26.	**k-f7**
7.	**r-e1**	14.	k-d4	21.	k-c7		**r-c8#**

Notes on **Solutions** to Problem **5**

Everyone used static board evaluation functions that value central squares more than edge squares; the ratio of **center:edge** preferences varied from **5:1** to $\mathbf{10^{10}}$**:1.**

RTB, DMO, and AMY's function gave a bonus to a board position if it split the set of possible Black King positions among two or more of the quadrants determined by White's Rook. NCR and MSK's function broke ties between board positions by favoring those for which the rectangle containing the set of possible Black King positions and one corner was largest.

Everyone tested for the possibility of capturing White's Rook or claiming stalemate before employing evaluation functions. DRF/MAI, RTB/DMO/AMY, NCR/MSK, ARR/DCH, and IAZ then evaluated the board positions resulting from each of the Black replies "OK," "Check ," and "Illegal," choosing the one that gave the best value. DAS/MES, CPP/CHT/WOL, and AFB implemented alpha-beta pruning to do 1½ move lookahead before deciding what move to make.

A couple of people mentioned that they'd planned to implement lookahead, but found it hard to beat their preliminary static board evaluation programs when they played White by hand, so they stuck with the simpler programs.

DRAMATIS PERSONÆ

The Professor

DEK Donald E. Knuth

The Teaching Assistant and Grader

c v w **Christopher J. Van Wyk**

The Teaching Assistant Assistant

IAZ Ignacio Zabala

The Students

AFB Anne **Beetem**
AMY Amy Plikerd
ARR Armando Rodriguez
CHC Charles **Clanton**
CHT Christopher Tong
CPP Charles Paulson
DAS David Smith
DCH Daniel Chapiro
DMO D. Michael Overmyer
DRF David Fuchs
DRR Doris Ryan
JEB James Boyce
JJF J. Jeffrey Finger
MAI Harry Mairson
MES Martin Schairer
MH Michael Hartstein
ML Michael Lowry
MSK Michael Kenniston
NCR Neil Rowe
RTB Richard Bagley
SOY Andras Solymosi
WDG William Gropp
WOL Pierre Wolper

The Visitors

BIA Beng t **Aspvall**
DRM Donald Morrison
OJD **Ole-Johan** Dahl