REMARK ON ALGORITHM 292 [S22]*
REGULAR COULOMB WAVE FUNCTIONS [Walter Gautschi, *Comm. ACM 9* (Nov. 1966), 793]
AND ON
REMARK ON ALGORITHM 292 [S22]
REGULAR COULOMB WAVE FUNCTIONS [Walter Gautschi, *Comm. ACM 12* (May 1969), 280]

W. J. Cody and Kathleen A. Paciorek (Recd. 8 Sept. 1969 and 8 May 1970)
Argonne National Laboratory, Argonne, IL 60439

* Work performed under the auspices of the US Atomic Energy Commission.

KEY WORDS AND PHRASES: Coulomb wave functions, wave functions, regular Coulomb wave functions
CR CATEGORIES: 5.12

The revised version of the procedure *Coulomb* was translated into IBM System/360 Algol and tested on an IBM S/360 Model 75 Computer. When $\eta > 12$ overflow problems were encountered in the generation of intermediate arrays. These were due to the smaller exponent range of the S/360, $-64 \leq exp \leq 63$. The following changes, while not completely eliminating the overflow probe ms, greatly alleviate them.

Insert **real** *scale*;
after **begin integer** *L*, *nu*, *nu*1, *mu*, *mu*1, *i*, *k*;

Insert *scale* := 16 ↑ (−57);
comment This value of *scale* is appropriate for the IBM S/360. On a machine with a different base and a different exponent range, say $\alpha \leq exp \leq \beta$, the value of *scale* should be *base* ↑ (6−β);
between **end**;
and *epsilon* := .5 × 10 ↑ (−*d*);

Change *lambda* [0] := *lmin* [0] := 1; *lambda* [1] := *omega-eta*; *sum* := *ro* × *exp* (*omega* × *ro*);
to *lambda* [0] := *scale*; *lmin* [0] := 1;
lambda [1] := (*omega-eta*) × *scale*;
sum := *ro* × *exp*(*omega* × *ro*) × *scale*;

Change *lmin* [*L*] := *Rra* [*L* − 1] × *lmin* [*L* − 1];
to **begin**
*t*1 := *Rra* [*L* − 1] × *lmin* [*L* − 1];
comment The following constant 5 ↑ (−10) is approximately 2 × *base* ↑ α/*scale*, where *base* is the base of the floating-point number system and $\alpha \leq exp \leq \beta$;
lmin[*L*] := **if** *abs*(*t*1) > 5 ↑ (−10) **then** *t*1 **else** 0
end;

Change *lam* [0] := −*r*1; *lam* [1] := 1;
to *lam* [0] := −*r*1 × *scale*; *lam* [1] := *scale*;

Change *lambda* [*L*] := *lmin* [*L*] + *t*1 × (*lam* [*L*] + *r*1 × *lmin* [*L*])
to *lambda* [*L*] := *lmin* [*L*] × *scale* + *t*1 × (*lam* [*L*] + *r*1 × *scale* × *lmin* [*L*])

Change *F*[0] := *sum*/(1+*s*);
to *F*[0] := *sum*/(*scale*+*s*);

The authors gratefully acknowledge the referee's helpful suggestions.

The policy concerning the contributions of algorithms to *Communications of the ACM* has been revised and was published in the August 1970 issue, page 513. Copies of "Algorithm Policy / Revised August 1970" will be mailed upon request.

Short Communications

PROGRAMMING TECHNIQUES

Comment on Bell's Quadratic Quotient Method for Hash Code Searching

Leslie Lamport
Applied Data Research, Inc., Wakefield, Massachusetts

Key Words and Phrases: hashing, hash code, scatter storage, calculated address, clustering, search, symbol table, keys, table look-up
CR Categories: 3.74, 4.9

In a recent paper [1], James R. Bell gave a method for resolving collisions in a hash coded table search which generalized the quadratic search method of W. D. Maurer [2]. However, he ignored an important anomaly of Maurer's method, as well as one singular case.

In the quadratic search method, table locations $k + ai + bi^2$ modulo p are examined sequentially for $i = 0, 1, 2, \cdots$; where p is the table size which is a prime number, k is the hash code of the entry's key, and a and b are constants with $b \not\equiv 0(p)$. The same table location is examined for $i = i_1$ and $i = i_2$, $i_1 \not\equiv i_2$, if and only if

$$k + ai_1 + bi_1^2 \equiv k + ai_2 + bi_2^2(p)$$

which is true if and only if

$$(i_2 - i_1)[a + b(i_1 + i_2)] \equiv 0.$$

This in turn is true if and only if

$$i_1 \equiv i_2 \quad \text{or} \quad i_1 + i_2 \equiv (p - a)/_p b,$$

where $/_p$ denotes division in the field of integers modulo p. It is easy to see from this that the procedure examines $(p + 1)/2$ different table locations, if $p > 2$. However, it will examine them all on its first $(p + 1)/2$ tries only if $a \equiv 0$. Therefore, for an efficient procedure we should take $a \equiv 0$. The search can then be stopped after $(p + 1)/2$ tries.

Bell's improvement of this procedure consists of letting a and b be pseudorandom functions of the entry's key. For his algorithm [1, p. 108],

$$a \equiv \text{“some fixed constant”} + Q/_p 2$$

$$b \equiv Q/_p 2,$$

where Q is a function of the entry's key, and $p > 2$. (Although Bell states that a is a constant, this is not true for the algorithm he describes.) The algorithm considers the search a failure (table "full" and entry not in it) when it returns to the first location examined. This occurs on the ith try, where $1 < i \leq p + 1$, $i - 1 \equiv (p - a)/_p b$. At that time, every other location which it has examined will have been examined twice. Clearly, half the table will be searched only if we replace the "fixed constant" by a number congruent to $-Q/_p2$. However, even if this is done, there is still the problem that when $Q \equiv 0$, only one table location is examined!

To correct these problems, replace steps (3) and (4) of Bell's algorithm with:

(3) Initialize A with C, where C is defined below.
(4) Increment A by $2Q$.

For this algorithm, we have $a = Q + C$, $b = Q$. We must then choose C so that $C \equiv -Q$ if $Q \not\equiv 0$ and $C \not\equiv -Q$ if $Q \equiv 0$. The algorithm will then search $(p + 1)/2$ locations if $Q \not\equiv 0$, and will search all p locations if $Q \equiv 0$.

The trouble with this algorithm is that it requires testing for $Q \equiv 0$, which means performing an extra division. A seemingly possible way out is to observe that if $(p - a)/_pb \equiv -j$, $b \not\equiv 0$, then the algorithm searches j fewer locations before it starts re-examining locations. We can then try to choose C so that we get j to be small, thereby examining nearly half the table before repeating. However, this requires that we make $C \equiv -(j + 1)Q$. There does not appear to be any simple algorithm for choosing a C satisfying this congruence for a small j when $Q \not\equiv 0$, and choosing $C \not\equiv -Q$ when $Q \equiv 0$.[1] It seems that the division is necessary.

The corrected version of Bell's algorithm still contains a gross inefficiency. For $Q \not\equiv 0$, it decided that the search is a failure after p tries, instead of the necessary $(p + 1)/2$ tries. This is easily corrected by changing the criterion for failure.

In summary, Bell's algorithm requires a correction which adds an extra division to the initialization procedure. This must be considered in evaluating its efficiency. Bell's table comparing the efficiency of his method with that of Maurer's indicates that this extra initialization cost is justified only if checking a single entry is a relatively time consuming operation.

References:

1. Bell, James R. The quadratic quotient method: a hash code eliminating secondary clustering. *Comm. ACM 13*, 2 (Feb. 1970), 107–109.
2. Maurer, W. D. An improved hash code for scatter storage. *Comm. ACM 11*, 1 (Jan. 1968), 35–38.

Reply by Bell. Before discussing Lamport's comment in detail, let us consider the correct observation on which it is based: Although any quadratic search (including quadratic quotient) hits half of the table entries, sometimes some entries are hit twice before others are hit once.

In other words, $K + ai + bi^2$ may not have maximum period for an arbitrary a and b. The author proves that forcing a to zero will guarantee maximal period.

A much simpler constraint is to let the constant in step (3) of the original algorithm be zero. Then

$$h_i(K) = R + (Q/2)i + (Q/2)i^2$$

and we first return to our original hash address when

$$R = R + (Q/2)i + (Q/2)i^2,$$

that is, when

$$i = -1 \quad \text{or} \quad i = 0 \quad \text{or} \quad Q = 0.$$

The first two cases state that $h(K)$ has a maximum periodicity. The third case is the degenerate one where the quotient is congruent to zero. We could use a division to spot the degenerate case. But by adopting the suggestion of paragraph 3 of Section 3c of the original article we can use

$$(Q \wedge \text{lowbitmask}) + 1$$

in lieu of Q to guarantee that this case does not occur.

Lamport has taken a more complicated approach.

[1] Note added in proof: In his reply below, Bell gives a simple method of choosing $j = 1$.

SCIENTIFIC APPLICATIONS

On the Number of Automorphisms of a Singly Generated Automaton

Zamir Bavel
University of Kansas, Lawrence, Kansas*

Key Words and Phrases: automata, finite automata, singly generated automata, automorphisms, generators, length of state, minimal-length generators, orbit.
CR CATEGORY: 5.22

1. Introduction

Weeg proved in [3] that the number of automorphisms of a strongly connected finite automaton divides the number of states of the automaton. In [1, Th. 6], the author generalized this result to finite singly generated automata by proving that the number of automorphisms of such an automaton A divides the number of generators of A. This brief note improves the latter result. The number of automorphisms of A is shown to divide the number of minimal-length generators of A.

The improvement is of practical value not only in the more general case of singly generated automata but also in the strongly connected case, for the number of states whose length is minimal is usually much smaller than the number of states of the automaton. The improvement is particularly striking when the number of generators (states, in the strongly connected case) is large but only one of them is of minimal length; in that case, the only automorphism is the identity. But without the present result it may be necessary to examine up to half the number of

* Department of Computer Science. This work was supported in part by the National Science Foundation under Grant GJ-639.