Efficient Timestamp Input and Output

Curtis E. Dyreson[1]
Richard T. Snodgrass[2]

TR 93-01

February 25, 1993

Abstract

In this paper we provide efficient algorithms for converting between the internal form of a timestamp, and various external forms, principally character strings specifying Gregorian dates. We give several algorithms that explore a range of time and space tradeoffs. Unlike previous algorithms, those discussed here have a constant time cost over a greatly extended range of timestamp values. These algorithms are especially useful in operating systems and in database management systems.

[1]Department of Computer Science
University of Arizona
Tucson, AZ 85721
`curtis@cs.arizona.edu`

[2]Department of Computer Science
University of Arizona
Tucson, AZ 85721
`rts@cs.arizona.edu`

Efficient Timestamp Input and Output

Contents

1 Introduction

In this paper we provide efficient algorithms for converting between the internal form of a timestamp, an integer, and various external forms, principally character strings specifying Gregorian dates, such as "January 1, 1993."

Algorithms that perform timestamp input and output are prevalent in most operating systems and database management systems. Efficiency has not been considered critical, in part because input and output performance is ultimately limited to the latency of I/O devices, which can be orders of magnitude slower than CPU speeds. As we'll demonstrate empirically later in this paper, input of the above-mentioned date using a commercially available routine takes approximately 3 msec, while output is substantially faster at 0.12 msec (on a 12-MIPS machine). To display the resulting string on a screen at 9600 baud takes much longer, about 13 msec.

The need for more efficient algorithms becomes apparent when considering *temporal databases* [Tansel, A. et al. 1993] that store time-varying information. Current input and output algorithms apply to restricted intervals of time (generally 1970 to 2038). If temporal databases are to be useful to astronomers, historians, archaeologists, and geologists, the time interval must be greatly lengthened. Unfortunately, the cost of these I/O algorithms is linear in the distance from the origin of the date to be input or output. So if we move the origin from 1970 to 1 A.D., inputting the above date will take 145 msec of CPU time!

In this paper we present a series of algorithms for both input and output. The final algorithms exhibit effectively constant performance over all of time (18 *Billion* B.C.E. to 18 Billion A.D.). By employing several algorithmic refinements and by trading space for time, we are able to get input down to about 8 μsec, and output down to about 22 μsec.

Section 2 presents a data model for timestamps, discussing both their semantics in terms of physical clocks and their representation as bit patterns. The next section argues that input and output are the most difficult of the timestamp operations to implement efficiently. We then develop output and input algorithms for Gregorian values in Sections 4 and 5, respectively. These algorithms are empirically compared with existing packages in Section 6. We briefly discuss the generality of the proposed approach by applying it to input and output of temporal constants using the Chinese lunar calendar.

Source code for all the algorithms is in the public domain. The programs and the test cases used to derive the results in this paper can be accessed via anonymous FTP from `cs.arizona.edu` in the subdirectory `tsql/fastIO`.

2 Timestamp Semantics and Representation

There are three basic temporal data types: *events*, *spans*, and *intervals* [Soo & Snodgrass 1992]. An event is an isolated instant in time. A span is an unanchored duration of time, while an interval is an anchored duration, the time between two events. In this paper we consider only event timestamps and operations on event timestamps. Span and interval operations are similar [Soo et al. 1992].

In the temporal database community, two basic time models have been proposed: the *continuous model*, in which time is viewed as being isomorphic to the real numbers, with each real number corresponding to a "point" in time, and the *discrete model*, in which time is viewed as being isomorphic to the integers [Clifford & Tansel 1985]. In the discrete model, the continuous time-line is partitioned into line segments. Each segment is called a *chronon* [Ariav 1986, Clifford & Rao 1987], which is an indivisible unit of time. We choose to use the discrete model.

The meaning of each chronon in our data model is given by the *time-line clock*. A time-line clock is a set of *physical clocks* coupled with some specification of when each physical clock is authoritative. Each chronon in a time-line clock is a chronon (or a regular division of a chronon) in an identified, underlying physical clock. A physical clock is a physical process coupled with a method of measuring that process. Although the underlying physical process is continuous, the physical clock measurements are discrete, hence a physical clock is discrete. A physical clock by itself does not measure time; it only measures the process. For instance, the rotation of the earth measured in solar days is a physical clock. The time-line clock switches from one physical clock to the next at a synchronization point. A synchronization point correlates two, distinct physical clock measurements.

Elsewhere we give the rationale behind a time-line clock composed of four physical clocks [Dyreson & Snodgrass 1993]. This clock extends from the beginning of time (specifically, the "Big Bang" [Hawking 1988], which occurred between 14 and 18 billion years ago) to 18 billion years into the future. The *time-line granularity* is one second. That is, every chronon in our time-line clock represents one second.

We adopt a particularly simple, yet adequate, timestamp format. The format is a 59-bit integer, with an additional sign bit, representing the number of seconds (chronons) from an arbitrary origin. Each timestamp bit pattern represents a unique chronon. This simple timestamp format facilitates efficient arithmetic and comparison operations, and is essential for output to and input from multiple calendars [Soo & Snodgrass 1992]. The algorithms we describe use Midnight January 1, 1970 A.D. as the timestamp origin, but we indicate for each algorithm how a different origin might be used. In C parlance, our representation is a `long long int`, which occupies 64 bits. Elsewhere we show how to exploit the remaining four bits [Dyreson & Snodgrass 1993].

3 Timestamp Operations

Liskov and Guttag have proposed a taxonomy of abstract data type operations [Liskov & Guttag 1986]. Applying this taxonomy to timestamps suggests four general categories of timestamp operations.

1. Operations that create timestamps without taking any timestamps as input, called *primitive constructors*.

2. Operations that create timestamps but take timestamps as input, called *constructors*.
3. Operations that modify timestamps, called *mutators*.
4. Operations that take timestamps as input and return other types, called *observers*.

Following this classification scheme, we have identified a comprehensive set of six primitive operations on event timestamps [Soo et al. 1992].

1. **create** — Create an event timestamp, the initial value of the timestamp is given as a calendar specific value, a primitive constructor.
2. **first** — Choose the earliest event from among a pair of events, a constructor.
3. **last** — Choose the latest event from among a pair of events, a constructor.
4. **shift** — Move an event by a specified duration, a mutator.
5. **precedes** — Decide if one event is before another, an observer.
6. **output** — Convert a timestamp value to a calendar specific value, an observer.

Note that **precedes** can be used to build the complete set of temporal predicates [Allen 1981] (e.g., $\mathbf{equals}(e_1, e_2)$ is equivalent to $\neg\mathbf{precedes}(e_1, e_2) \wedge \neg\mathbf{precedes}(e_2, e_1)$).

Efficient implementation of an event abstract data type depends on efficient implementation of these six functions. Some of these functions are hard to implement efficiently, while others are straightforward. In particular, **precedes** is a 64-bit comparison, **first** and **last** are trivial given **precedes**, and **shift** is but a 64-bit addition operation. In the rest of this paper, we focus on the two remaining functions, which are difficult to implement.

Create is challenging to implement since it must convert a calendar specific value (e.g., a Gregorian calendar date) to a timestamp value, the number of seconds from the origin. We call this conversion task *input*. We shall see that doing input efficiently is complex. For the same reasons, implementing **output** is hard. We develop algorithms that efficiently perform Gregorian calendar input and output. We anticipate that Gregorian calendar input and output will be the most common kind of input and output and are the topics of the next two sections.

4 Gregorian Output

Output converts the time stored in a timestamp to a *calendar specific representation*, that is, an array of pertinent calendar values [Soo & Snodgrass 1992]. The Gregorian calendar representation, for instance, has array values for the year, month, and day, whereas a business calendar representation has array values for the fiscal year, quarter, and work day. Constructing a character string from such an array is straightforward. With an auxiliary array of month names, the three integer

```
procedure SECONDS_TO_GREG(in seconds_in : integer; out year, out month, out day : integer);
    const
        seconds_between_origins : integer = 62167132800; { Origin is 1970 }
    var
        seconds : integer; { 64 bit integer }
    begin
        seconds := seconds_in + seconds_between_origins; { Adjust generic origin }
        if seconds > 0 then
            OUTPUT_AN_AD_DATE(seconds, year, month, day)
        else
            OUTPUT_A_BC_DATE(seconds, year, month, day)
    end; { SECONDS_TO_GREG }
```

Figure 1: Interface to output algorithm

array values 1, 1, and 1993 (denoting month, day in month, and year, respectively) can be mapped into the string "January 1, 1993."

We describe a series of algorithms, each of which outputs a timestamp in the Gregorian calendar representation. We anticipate that Gregorian calendar output will be the most common kind of output. We first give a "brute force" algorithm that is space efficient, but time inefficient, and follow with algorithms that improve time efficiency but incur modest space overheads. We describe two such algorithms, one requiring 8K bytes of main memory, and the other needing 22K.

The algorithms we give output only A.D. dates. We omit similar algorithms which output B.C.E. dates for expository brevity. The B.C.E. output algorithms are essentially the same as the A.D. algorithms, except that different tables are used. When calculating space costs, however, we only include the size of the tables needed for outputting A.D.; outputting B.C.E. dates doubles the space costs.

The interface to the output routine is given in Figure 1. The interface merely aligns the timestamp origin (Midnight January 1, 1970 A.D.) with the Gregorian calendar origin (Midnight January 1, 1 A.D.) and determines whether the B.C.E. or A.D. output algorithm need be invoked. Note that a different timestamp origin may be used with no loss of efficiency by simply redefining the *seconds_between_origins* constant.

4.1 Brute Force Approach

The brute force algorithm, shown in Figure 2, converts a timestamp, representing a positive number of seconds from the origin, to a Gregorian year, month, and day. We assume, initially, that the timestamp does not count leap seconds. Hence, each day contains exactly 86400 seconds. We consider code adjustments to handle leap seconds in Section 4.4.

Computing the Gregorian date corresponding to some number of seconds from the origin is complex because, while every day contains the same number of seconds, not every year contains the same number of days. Gregorian years are 365 days long except leap years which are 366. Leap years are years that are evenly divisible by 4, but not divisible by 100, unless the year is divisible by 400 (this is the common leap year rule, although other rules have been proposed [Aveni 1989]). For example, 1900 is not a leap year while 2000 is a leap year.

The output algorithm is based on the observation that the Gregorian calendar repeats itself in four hundred year periods. That is, every 400 year period from 1 A.D. is the same number of days, $365 \times 400 + 97 = 146097$ days (there are 97 leap days every 400 years) to be exact. We term this 400 year period a *greg*. The brute force algorithm calculates the number of gregs between 1 A.D. and the date to be output and subtracts the appropriate number of seconds from the timestamp. This reduces the task of outputting a date in the range 1 A.D. to 18 Billion A.D. to outputting a date in the range 1 A.D. to 401 A.D. (We are hedging our bets that we can efficiently output a date in the range 1 A.D. to 401 A.D.) For example, to output a timestamp corresponding to Midnight July 2nd, 1970 A.D., we observe that there are three gregs prior to 1970. This reduces the output task to converting the timestamp corresponding to Midnight July 2nd, 370 A.D.

The output algorithm pinpoints a specific year in the 400 year range, 1–401 A.D., by counting year by year. Each year counted decrements the cumulative remaining seconds by the number of seconds in that year (leap years contain more seconds than other years). The year by year count is terminated when the remaining seconds are fewer than the number of seconds in a year. This technique not only calculates a specific year in the 400 year range, it also isolates the number of seconds from the start of that year to the date being translated. The latter value is used to determine the day and month. For brevity, we omit the straightforward "special case" code corresponding to the functions *day_to_month*, *day_to_day*, and *leap_year*.

4.2 Trading Space for Time

The algorithm given above determines whether a year is a leap year for each year in the 1–401 A.D. range by calling the *leap_year* function. Since there are only 400 possible inputs to the *leap_year* function, the function can be precomputed and stored in a table, avoiding some on the fly computation. Similarly, the *day_to_month* and *day_to_day* functions can be precomputed and stored in a table. These tables occupy 528 bytes.

Another optimization improves the efficiency of pinpointing the year in the 1–401 A.D. range and determining how many days into that year the reduced timestamp represents. The brute force algorithm uses a loop, which is iterated a minimum of 0 times and a maximum of 400 times. The optimized algorithm makes an initial, optimistic guess at the year and days into that year and then corrects the guess. The initial guess ignores leap years and presumes that every year is exactly 365 days. The initial guess overestimates the actual year and day count. The correction to the initial guess factors in leap days. The cumulative number of prior leap days for each year in the 1–401 A.D. year range are precomputed and stored in the *leap_days_in_years* table. Since there are 97 leap days in a greg, the initial guess can overestimate the year count by at most 1 year, and the day count by at most 97 days. The correction decrements the year and day count as needed.

```
procedure OUTPUT_AN_AD_DATE(in seconds_in : integer; out year, out month, out day : integer);
    const
        how_many_seconds_in_year: array [boolean] of 365..366;
        seconds_in_day : integer = 86400;
        seconds_in_400_years : integer = 146097 × seconds_in_day;
    function leap_year(integer): boolean;
    function day_to_month(boolean, integer) : 1..12;
    function day_to_day(boolean, integer) : 1..31;
    var
        years, days, seconds : integer;
    begin
        years ← ((seconds_in div seconds_in_400_years) × 400) +1;
        seconds ← seconds_in mod seconds_in_400_years;
        while seconds > seconds_in_year do
            years ← years + 1;
            seconds ← seconds − how_many_seconds_in_year[leap_year(years)];
        days ← seconds div seconds_in_day;
        month ← day_to_month(leap_year(years), days);
        day ← day_to_day(leap_year(years), days)
    end; { OUTPUT_AN_AD_DATE }
```

Figure 2: The brute force output algorithm

The improved algorithm using the optimized method of pinpointing the year and the technique of caching function results is shown in Figure 3.

4.3 Further Optimizations

The problem with the second algorithm is that it is riddled with expensive division, multiplication, and modulus operations. All of these operations can be optimized away.

We focus on the code to determine how many complete gregs and how many days into the next greg a timestamp represents. In the second algorithm, the code that calculates these values performs two 64 bit integer divisions and one 64 bit modulus. We can replace these operations with a process of repeated additions. The trick is to examine the timestamp bit-by-bit. Since the timestamp is a binary number, the i^{th} timestamp bit represents 2^{i-1} seconds. Each 2^{i-1} quantity of seconds corresponds to some number of gregs, leftover days, and leftover seconds. The leftover days are a quantity of days that don't quite add up to one greg (less than 146097 days). The leftover seconds are a quantity of seconds that are less than the number of seconds in a day (86400 seconds). For example, the fortieth bit in the timestamp represents 2^{39} seconds which corresponds to 43 gregs, 80743 leftover days, and 44288 leftover seconds. The gregs, leftover days, and leftover seconds corresponding to each bit are precomputed and stored in three arrays. As each bit in the timestamp is tested, a running sum of each quantity: gregs, leftover days, and leftover seconds, is

```
procedure OUTPUT_AN_AD_DATE(in seconds_in : integer; out year, out month, out day : integer);
    const
        seconds_in_day : integer = 86400;
        seconds_in_year : integer = seconds_in_day × 365;
        seconds_in_400_years : integer = 146097 × seconds_in_day;
        leap_year : array[0..399] of boolean;
        how_many_days_in_year : array[boolean] of 365..366;
        leap_days_in_years : array[0..400] of 0..97;
        day_to_month : array[boolean, 0..365] of 1..12;
        day_to_day : array[boolean, 0..365] of 1..31;
    var
        years, days, gregs : integer;
    begin
        { calculate gregs }
        gregs ← (seconds_in div seconds_in_400_years);

        { calculate year in greg }
        days ← (seconds_in mod seconds_in_400_years) div seconds_in_day;
        years ← days div 365; { make initial guess }
        days ← (days mod 365) − leap_days_in_years[years];
        if (days < 0) then { apply correction to guess }
            years ← years − 1;
            days ← days + how_many_days_in_year[leap_year[years]];
        year ← (400 × gregs) + years + 1;

        { calculate month and day in year}
        month ← day_to_month[leap_year[years], days];
        day ← day_to_day[leap_year[years], days]
    end; { OUTPUT_AN_AD_DATE }
```

Figure 3: A greg counting output algorithm

maintained. We could maintain a sum of the number of years corresponding to each greg rather than a sum of the gregs, but this would require using a 64 bit integer and 64 bit addition operations. We found that it is quicker to use a 32 bit counter and then multiply the number of gregs counted by 400 (using 3 shifts and 3 additions).

To improve the efficiency of the bit counting technique, four bits are counted each iteration instead of just a single bit. The loop treats the first four bits as the first group, the next four bits as the second group, etc. Since a timestamp has 59 significant bits, there are at most 15 groups. Each group corresponds to a value in the range 0–15. For each group, all 16 possibilities for the number of gregs, leftover days, and leftover seconds are stored in separate tables, each table is $15 \times 16 \times 4 = 960$ bytes in size.

After each group of four bits in the timestamp has been tested, the leftover seconds and days must be reapportioned. The leftover seconds are reapportioned into days. To compute the number of days represented by the leftover seconds, we could divide the leftover seconds by the number of seconds in a day. But there can be a maximum of 1,700,000 leftover seconds (this is the sum of all the leftover seconds). Hence, the leftover seconds constitute at most $(1,700,000 \textbf{ div } 86,400) = 19$ days. We can avoid a division by storing all 20 outcomes (zero days, one day, ..., nineteen days) in a binary search tree of depth 5 with 20 leaves and 31 interior nodes. Each leaf represents a number of days to add and a number of seconds to subtract from the leftover seconds (assuming this value is needed later in the algorithm). The interior nodes direct the search to the appropriate leaf using the leftover seconds as the key. The leftover days are reapportioned into gregs using a binary search tree as well. There can be a maximum of 2,900,000 leftover days (which represents at most 19 gregs).

The other expensive operations in the second algorithm are a division and modulus by a constant (365) of a value in the range 0–146097. We can replace these operations using the binary search technique discussed above. In this case, the binary search tree has 400 leaves. The rest of the second algorithm remains unchanged in the new, optimized version. The complete algorithm is shown in Figure 4. A key feature of this algorithm is that few 64-bit arithmetic and comparison operations are needed as most of these operations have been "optimized" away. In the end, only one addition, one multiplication, one exclusive-or, and one right shift remain.

4.4 Leap Seconds

Universal Coordinated Time (UTC) relates the atomic clock to the Earth's rotational clock. UTC leap seconds are used to synchronize the atomic clock and the Earth's varying rate of rotation. If the Earth's rotation slows with respect to the atomic clock, a second is added to UTC. If the rotation quickens, a second is subtracted. Preference is given to July 1 and January 1 to add or subtract seconds. A total of twenty-seven leap seconds changes (all additions) have been made since the adoption of the leap second system in 1972. The most recent leap second added was just prior to Midnight, July 1, 1992. In future, more leap seconds may be added (or removed).

The algorithms given in previous sections output timestamps that do not count leap seconds. If leap seconds can be removed from a timestamp that counts them, these output algorithms still suffice. Towards this end, we use a table of three columns and twenty-seven rows (one row for each leap second change). The first column of each row in the table is the timestamp value when the leap second was added (or removed). The second column is the cumulative number of leap seconds added and subtracted prior to the timestamp value in the first column (note that the values in this column could differ from the index). Finally, the third column contains the Gregorian array values corresponding to the timestamp value given by the first column. We assume the table is sorted by the first column.

To output a timestamp that counts leap seconds, the leap second table described above is searched with the timestamp as the search key (using binary search). If an exact match is found, the Gregorian array values corresponding to the timestamp are retrieved and outputted (since the output algorithms cannot produce these values). If an exact match is not found, the closest timestamp value greater than the timestamp to be output is selected. The cumulative leap second

```
procedure OUTPUT_AN_AD_DATE(in seconds_in : integer; out year, out month, out day : integer);
    const
        leap_year : array[0..399] of boolean;
        day_to_month : array[boolean, 0..365] of integer;
        day_to_day : array[boolean, 0..365] of integer;
        greg_count : array[1..15,1..15] of integer;
        leftovers_days : array[1..15,1..15] of integer;
        leftovers_seconds : array[1..15,1..15] of integer;
        seconds_reapportionment : array[1..20] of integer;
        greg_modulus : array[1..20] of integer;
        days_reapportionment : array[1..20] of integer;
        year_reapportionment : array[1..400] of integer;
        days_within_year : array[1..400] of integer;
    var
        gregs, years, seconds, days, year_residue : integer;
    begin
        { calculate gregs }
        gregs ← days ← seconds ← 0;
        for i ← 1 to 15 do
            j ← value of current group of 4 bits;
            if j > 0 then
                gregs ← gregs + greg_count[i,j];
                seconds ← seconds + leftovers_seconds[i,j];
                days ← days + leftovers_days[i,j];
        days ← days + binary_search(seconds_reapportionment, seconds);
        gregs ← gregs + binary_search(days_reapportionment, days);

        { calculate year in greg and offset within that year }
        days ← days − binary_search(greg_modulus, days);
        years ← binary_search(year_reapportionment, days);
        year_residue ← days − binary_search(days_within_year, days);

        { calculate year, month and day }
        year ← years + (gregs × 400) + 1;
        month ← day_to_month[leap_year[years], year_residue];
        day ← day_to_day[leap_year[years], year_residue]
    end; { OUTPUT_AN_AD_DATE }
```

Figure 4: A greg counting output algorithm without expensive arithmetic operations

```
procedure SECONDS_TO_HOURS(in seconds_in : integer; out hour, out minute, out second : integer);
    var
        seconds_residue : integer;
    begin
        hour ← seconds_in div 3600;
        seconds_residue ← seconds_in mod 3600;
        minute ← seconds_residue div 60;
        second ← seconds_residue mod 60
    end; { SECONDS_TO_HOURS }
```

Figure 5: A brute force algorithm to count temporal divisions of a day

value associated with the closest timestamp value greater than the output timestamp is subtracted from the output timestamp. The subtraction removes the leap seconds from the timestamp, and the output operation can proceed as before. For instance, given a timestamp value that corresponds to Noon February 23, 1992 A.D., the table search does not find an exact match since this is not a time when a leap second was added. The closest leap second change after this date was the addition of the 27^{th} leap second just before Midnight July 1, 1992 A.D. Twenty-six seconds are subtracted from the timestamp value and the output operation proceeds as before. The search table technique for leap seconds can be added to *SECONDS_TO_GREG*, making the interface routine marginally more complex.

4.5 Hours, Minutes, and Seconds

Sometimes, the hour, minute, and second of a given timestamp value is output along with the year, month, and day. These values can be easily computed from the number of seconds remaining after counting off the number of seconds in the year, month, and day. (In the final output algorithm, this value is the number of seconds remaining after the leftover seconds are reapportioned into days.)

We present two methods of computing the hour, minute, and second. The first method, shown in Figure 5 uses two divisions and two modulus operations. The second method, shown in Figure 6 uses the binary search technique replacing each expensive division and modulus operation with an efficient table lookup. The second method is based on the observation that the expensive arithmetic operations have a limited number of possible outcomes (e.g., there can be at most 24 hours).

4.6 Time and Space Analysis

We implemented the three output algorithms in the C programming language. We call the first algorithm (Figure 2) the *brute force* algorithm. The second algorithm, with an improved counting technique (Figure 3), is called the *slender* algorithm. The final algorithm, using the special bit counting technique and several binary searches (Figure 4), is labeled the *full* algorithm. We

```
procedure SECONDS_TO_HOURS(in seconds_in : integer; out hour, out minute, out second : integer);
    const
        hours_reapportionment : array[1..24] of integer;
        hour_modulus : array[1..24] of integer;
        minutes_reapportionment : array[1..60] of integer;
        minutes_modulus : array[1..60] of integer;
    var
        seconds_residue : integer;
    begin
        hour ← binary_search(hours_reapportionment, seconds_in);
        seconds_residue ← seconds_in − hour_modulus[hour];
        minute ← binary_search(minutes_reapportionment, seconds_residue);
        second ← seconds_residue − minutes_modulus[minute]
    end; { SECONDS_TO_HOURS }
```

Figure 6: A table-based algorithm to count temporal divisions of a day

compiled the algorithms using the GNU C compiler, version 2.0, with compiler optimization fully enabled. We chose this compiler in part because it supports a 64 bit integer type (`long long`). All the tests were performed in a controlled environment on a dedicated Sun-4 IPC.

Each algorithm was tested on 400 different output dates (Midnight January 1st of each year from 1970–2370 A.D.). A single test consisted of 10 separate runs, where each run executed the output algorithm 10000 times to avoid any internal clock sampling errors. Figure 7 plots the run with the cost for each date. Both the slender and full algorithms are fairly stable throughout this range. The speed of the slender algorithm varies somewhat due to small perturbations in the speed of the division operation on a Sun-4 IPC architecture (division is microcoded as repeated subtraction). The speed of the full algorithm also varies slightly, it does fewer additions in the main loop depending upon which groups of four bits are "on" in the input timestamp. But the time cost of both the slender and full algorithms is mostly independent of the date to output, each is stable over the entire timestamp range (14 Billion B.C.E. to 18 Billion A.D.). The speed of the brute force approach, however, depends on the output date; dates in years just after a greg boundary are output much quicker than dates in years just prior to a greg boundary. This is caused by the loop in the brute force algorithm that counts, year by year, from the greg boundary to the output year.

Figure 8 gives a space and time comparison of the three algorithms. The space cost (in bytes) of each algorithm is shown first. Although there is a dramatic difference is space between the three algorithms, even the full algorithm has modest space costs (22KB). The time cost (accurate to less than a μsec) shows average execution speed over the 400 year test period. It also shows the minimum and maximum speeds as well as the dates that caused the minimum and maximum. Note that cost of the slender and full algorithms do not vary much. These tables illustrate the space vs. time tradeoff that we exploited in the design of these algorithms. The user can choose the algorithm that best meets their needs.

We independently tested the algorithms to output the hour, minute, and second of a particular

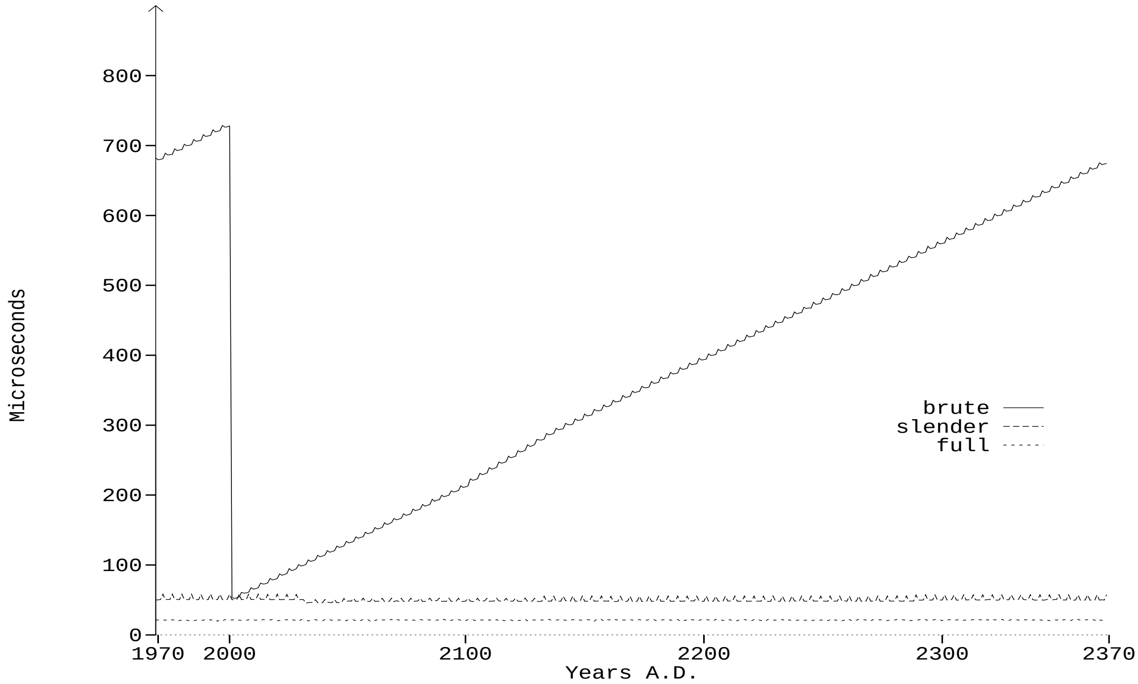

Figure 7: A plot of output execution speed over the first 400 years after the origin

Space (in bytes)

Algorithm	*Code Size*	*Data Size*	*Total Size*
Brute	752	96	848
Slender	504	7856	8360
Full	11072	11040	22112

Time (in μsec)

Algorithm	*Avg. Time*	*Min. Time*	*Min. Date*	*Max. Time*	*Max. Date*
Brute	400	54	2000 A.D.	740	1999 A.D.
Slender	51	51	—	51	—
Full	22	22	—	22	—

Figure 8: A comparison of costs for the three algorithms

Space (in bytes)

Algorithm	*Code Size*	*Data Size*	*Total Size*
Brute	72	0	72
Full	1424	336	1760

Time (in μsec)

Algorithm	*Avg. Time*	*Min. Time*	*Min. Date*	*Max. Time*	*Max. Date*
Brute	8.3	4.8	Midnight	9.8	11:59:59 P.M.
Full	3.4	3.4	—	3.4	—

Figure 9: A comparison of costs for the hour, minute, and second output algorithms

date using the same tests described above, except that the range of tested input times was the start of each minute during a day (3600 tests in total). We term the first algorithm, with expensive arithmetic operations, the *brute force* algorithm, and the second algorithm, with the expensive arithmetic operations replaced by binary searches and table-lookups, the *full* algorithm. A time and space comparison of the algorithms is given in Figure 9.

5 Gregorian Input

The second operation that is difficult to perform efficiently is **input**. Input converts a calendar specific representation, an array of pertinent Calendar values, into a timestamp. We make the same assumptions for input as we did for output concerning both the calendar specific representation and the timestamp format (e.g., a timestamp is a 64 bit signed integer recording the number of seconds from Midnight January 1, 1970). We focus on Gregorian calendar input since we anticipate that it will be the most common kind of input. We describe a simple, "brute force" input algorithm that is space efficient but time inefficient, and follow with algorithms that trade time for space. As we did for output, we give algorithms that input only A.D. dates, omitting algorithms that input B.C.E. dates. The B.C.E. input algorithms are essentially the same as the A.D. algorithms, except that different tables are used. When calculating space costs we only include the size of the tables needed for translating all A.D. dates; to input B.C.E. dates simply double the space costs. The interface to the input routine is given in Figure 10. The interface determines which input algorithm is needed and aligns the Gregorian calendar origin with the timestamp origin. Note that a different timestamp origin may be chosen with no loss of efficiency by redefining the *seconds_between_origins* constant.

5.1 Brute Force Approach

An input Gregorian date is three values: a year, a month, and a day. The task of input is to calculate the number of seconds between the origin and the input date. The brute force input algorithm, shown in Figure 11, uses a counting technique. The algorithm counts year by year adding the number of seconds in each year to a running total, starting with 1 A.D. and finishing

```
function GREG_TO_SECONDS(in year, month, day : integer) : integer;
    const
        seconds_between_origins : integer = 62167132800; { Origin is 1970 }
    var
    begin
        if year > 0 then
            return INPUT_AN_AD_DATE(year, month, day) − seconds_between_origins
        else
            return INPUT_A_BC_DATE(year, month, day) − seconds_between_origins
    end; { GREG_TO_SECONDS }
```

Figure 10: Interface to input algorithm

with the input year. Not every year has the same number of seconds; leap years have 86400 more seconds than other years. Once the years have been counted, the number of seconds from the start of the year to the start of the month and to the start of the day within that month are counted and added to the year count. A table of precomputed values stores the number of seconds from the start of a year to a particular month. The table has entries for both leap years and nonleap years since the number of seconds from the start of the year differs for the months past February depending on the type of year. All table entries are 32-bit values; the value returned is a 64-bit integer.

5.2 Trading Space for Time

The brute force algorithm counts each year between the Gregorian origin and the input year. But since there are the same number of seconds in every 400 year period from the origin (in every greg), it is more efficient to count in gregs, 400 years at a time. The optimized algorithm, shown in Figure 12, counts the number of gregs between the Gregorian origin and the input year. For time efficiency, the cumulative number of seconds to the start of the year for each year in the range 1–401 A.D. is precomputed and stored in an table. A table lookup retrieves the number of seconds between the nearest earlier 400 year boundary and the input year.

As was done in the output algorithm, the *leap_year* function is replaced in the optimized input algorithm by a table of precomputed results avoiding some on the fly computation.

5.3 Further Optimizations

The problem with the algorithm in Figure 12 is that it must perform two multiplications, one division, and one modulus. We can replace these expensive operations using the bit counting technique described in Section 4.3.

The input year is a binary number. Each bit in the input year represents a number of gregs

```
function INPUT_AN_AD_DATE(in year, month, day : integer) : integer;
    const
        seconds_in_day : integer = 86400;
        how_many_seconds_in_year: array [boolean] of integer;
        month_to_seconds: array [boolean, 1..12] of integer;
    function leap_year(integer): boolean;
    var
        seconds : integer;
    begin
        { Convert year to seconds }
        seconds ← 0;
        for j ← 1 to (year − 1) do
            seconds ← seconds + how_many_seconds_in_year[leap_year(j)];

        { Convert month and day to seconds }
        return seconds +
            month_to_seconds[leap_year(year), month] +
            (day − 1) × seconds_in_day
    end; { INPUT_AN_AD_DATE }
```

Figure 11: The "brute force" input algorithm

```
function INPUT_AN_AD_DATE(in year, month, day : integer) : integer;
    const
        seconds_in_day : integer = 86400;
        seconds_in_400_years : integer = 146097 × seconds_in_day;
        leap_year: array [1..400] of boolean;
        year_to_seconds: array [1..400] of integer;
        month_to_seconds: array [boolean, 1..12] of integer;
    var
        years : integer;
    begin
        years ← (year mod 400) + 1;
        return (year div 400) × seconds_in_400_years +
            year_to_seconds[years] +
            month_to_seconds[leap_year[years], month] +
            (day − 1) × seconds_in_day
    end; { INPUT_AN_AD_DATE }
```

Figure 12: An optimized input algorithm

and a number of leftover years. More specifically, the i^{th} bit represents (2^{i-1} **div** 400) gregs and (2^{i-1} **mod** 400) leftover years. For example, the tenth bit represents (2^9 **div** 400) = 1 greg and (2^9 **mod** 400) = 112 leftover years. The input year is examined bit-by-bit and a running sum of the gregs (actually, the number of seconds corresponding to those gregs) and leftover years is maintained. Again, instead of counting a single bit-by-bit, groups of four bits are counted. Once the bit examination is complete, the leftover years are reapportioned into gregs using the binary search technique discussed previously.

The bit counting technique eliminates all the expensive operations save one, the multiplication to determine the number of seconds corresponding to the input day. However, since there can be at most 31 days in a month, it it possible to precompute the results and store them in a table. The multiplication is replaced by a table lookup. The new algorithm, incorporating both of the optimizations described in this section, is shown is Figure 13.

5.4 Leap Seconds

The input algorithms given in previous sections input timestamps without leap seconds. We recommend leaving these algorithms unchanged and modifying only *GREG_TO_SECONDS* to accommodate leap seconds. Leap seconds can easily be added to a timestamp that has been already been converted by an input algorithm.

Towards this end, the user should create a two column table of timestamp changes. The first column lists the timestamp value (counting leap seconds) when a leap second is added or removed. The second column is the cumulative leap seconds to the value in the first column. The table should be searched using the timestamp value produced by the input algorithm (a standard binary search technique suffices). The cumulative leap seconds associated with the latest prior timestamp is added to the input timestamp. For instance, given a timestamp value that corresponds to Noon August 23, 1992, the latest prior leap second change was the addition of the 27th leap second just before Midnight July 1, 1992. To obtain the timestamp value corrected for leap seconds, 27 seconds must be added to the value produced by the input algorithm.

5.5 Hours, Minutes, and Seconds

Sometimes, the hour, minute, and second of a given Gregorian calendar value is input along with the year, month, and day. We describe a method of computing the number of seconds corresponding to a given hour, minute, and second. Multiply the hour by the number of seconds in an hour, and the minute by the number of seconds in a minute. Add the results of the multiplications to the input second. Since the multiplications are by a constant they can be replaced by two left shifts and a subtraction, i.e., $60 \times\ h\ =\ (h\ <<\ 6)\ -\ (h\ <<\ 2)$.

```
function INPUT_AN_AD_DATE(in year, month, day : integer) : integer;
    const
        greg_count: array [1..9,1..15] of integer;
        leftovers_years: array [1..9,1..15] of integer;
        year_reapportionment: array [1..14] of integer;
        year_modulus: array [1..14] of integer;
        leap_year: array [1..400] of boolean;
        year_to_seconds: array [1..400] of integer;
        month_to_seconds: array [boolean, 1..12] of integer;
        day_to_seconds: array [1..31] of integer;
    var
        seconds, years, i, j : integer;
    begin

    { Convert years to seconds }
        seconds ← years ← 0;
        for i ← 1 to 9 do { at most 36 significant bits in year }
            j ← value of current group of 4 bits;
            if j > 0 then
                seconds ← seconds + greg_count[i,j];
                years ← years + leftovers_years[i,j];

    { Reapportion leftovers }
        seconds ← seconds + binary_search(year_reapportionment, years);
        years ← years + binary_search(year_modulus, years);

    { Convert months and days to seconds }
        return seconds +
            year_to_seconds[years] +
            month_to_seconds[leap_year[years], month] +
            day_to_seconds[day]
    end; { INPUT_AN_AD_DATE }
```

Figure 13: An input algorithm that avoids expensive arithmetic operations

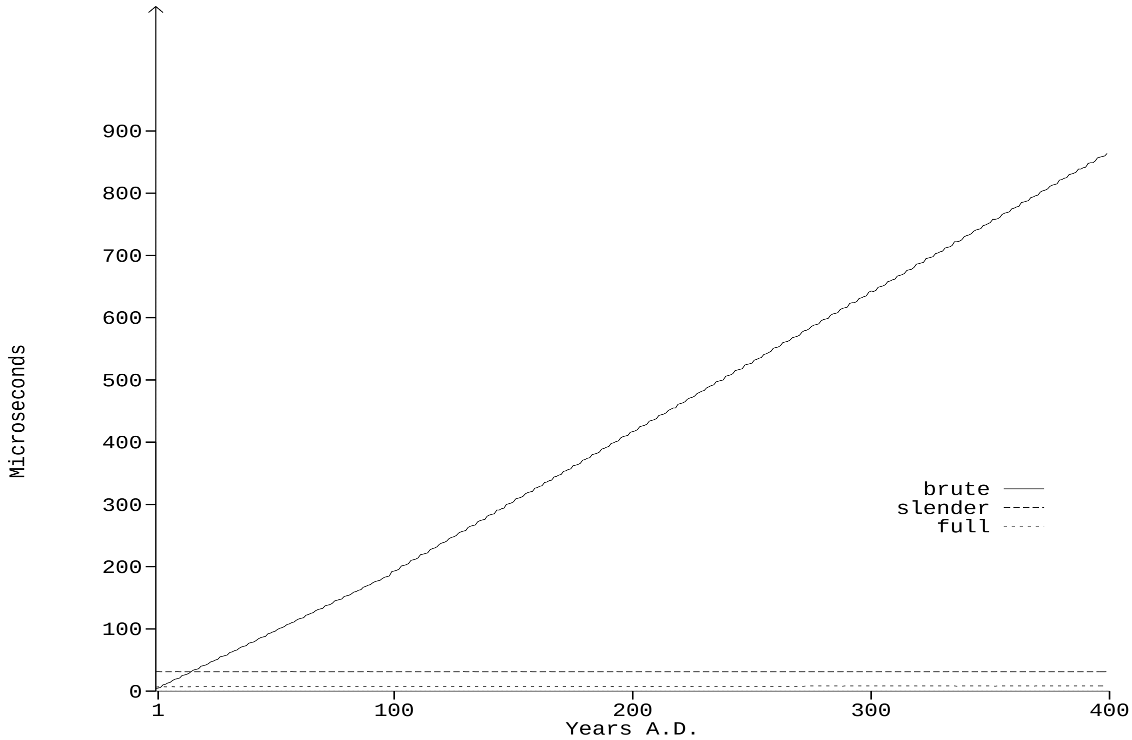

Figure 14: A plot of the input execution speed over the first 400 years

5.6 Time and Space Analysis

We implemented the three input algorithms in the C programming language. We call the first algorithm the *brute force* algorithm, the second algorithm, with an improved counting technique, the *slender* algorithm, and the final algorithm, using the bit counting technique, the *full* algorithm. We performed the same tests described in Section 4.6. All the tests were performed in a controlled environment on a dedicated Sun-4.

Each algorithm was tested on 400 different input dates (Midnight January 1st of each year from 1–401 A.D.). Figure 14 plots the run with the cost for each date. Both the slender and full algorithms are relatively stable throughout this range. The speed of the slender algorithm does vary somewhat due to perturbations in the speed of the division operation on a Sun-4 IPC architecture. The speed of the full algorithm also varies slightly, it does fewer additions in the main loop depending upon which groups of four bits are "on" in the input year. But the time cost of both the slender and full algorithms is mostly independent of the input date, each is stable over the entire timestamp range (14 Billion B.C.E. to 18 Billion A.D.). The speed of the brute force approach, however, depends on the input date; the time taken increases linearly from 1 A.D. This is caused by the loop in the brute force algorithm that counts, year by year, from the 1 A.D. to the input year. Hence the speed of the brute force algorithm is proportional to the distance of the input date from 1 A.D.

Figure 15 gives a space and time comparison of the three algorithms. The tables illustrate the space-time tradeoff we exploited in the design of the three algorithms: as space costs increase, time costs decrease.

Space (in bytes)

Algorithm	*Code Size*	*Data Size*	*Total Size*
Brute	424	104	528
Slender	272	3704	3976
Full	1048	5728	6776

Time (in μsec)

Algorithm	*Avg. Time*	*Min. Time*	*Min. Date*	*Max. Time*	*Max. Date*
Brute	420	5	1 A.D.	870	400 A.D.
Slender	31	31	—	31	—
Full	8	7	1 A.D.	9	400 A.D.

Figure 15: A comparison of costs for the three algorithms

6 Existing Input and Output Packages

There are three widely-available public domain or commercial packages that perform Gregorian calendar input and output. All of these packages use a 32-bit timestamp that records the number of seconds from Midnight January 1, 1970 (since the sign bit is not used, this yields a range of approximately 68 years). In the Unix operating system, the `timegm()` function performs input, converting a Gregorian specific representation (a `tm` struct) into a timestamp value, while `gmtime()` handles output. The Localtime public domain package offers a replacement for many Unix time functions including `timegm()` and `gmtime()`. The Free Software Foundation also offers a public domain version of both `gmtime()` and `timegm()` (called `mktime()`) as part of the GNU project. We will call these three packages, Unix , Localtime , and Gnu , respectively.

We note that all three packages are from operating systems environments. While database management systems also need to perform Gregorian calendar input and output, database timestamp typically store the Gregorian calendar values directly (e.g., SQL2's `datetime` timestamp stores the year, month, day, hour, minute, and second of a Gregorian date). Hence, databases typically do not perform input and output as we have defined them. However, typical database-style timestamps are space-inefficient and their formats make arithmetic and comparison operations as well as input and output in non-Gregorian calendars difficult and costly [Dyreson & Snodgrass 1993].

Although `timegm()` performs Gregorian calendar input as we have defined it, the function also checks the input parameters and allocates a `tm` struct in memory. Similarly `gmtime()` does more than the output algorithms described in Section 4, such as determining the day of the week.

We performed tests on these input and output functions similar to those reported in Sections 4.6 and 5.6. Because of the limited timestamp range, we only tested over a range of 68 years. A plot of the execution of each input function is given in Figure 16 while the execution of each output function is plotted in Figure 17. The figures show that the Gnu package consistently outperformed the other packages. The figures also show, for every function, a linear dependence between the speed of execution and the distance of the tested date from the origin. This is the

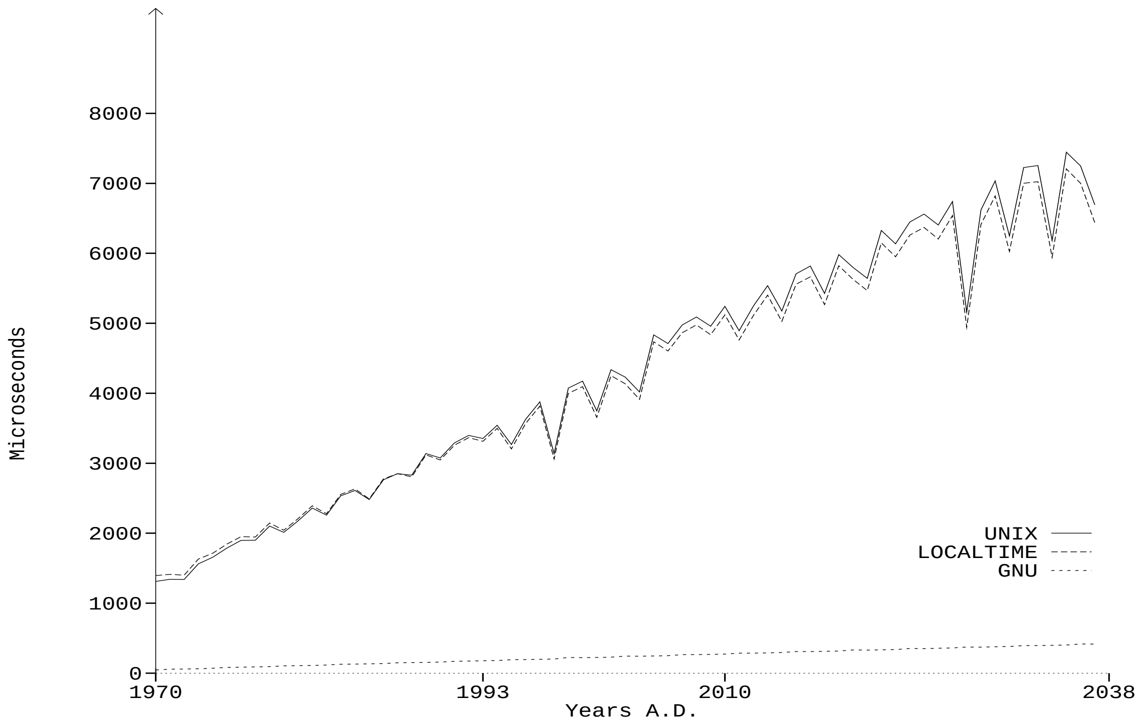

Figure 16: A plot of input execution times over the range of a UNIX timestamp

signature of the brute-force algorithms. Recall that the brute-force algorithms are space efficient. It is certainly unsurprising to find, in an operating systems environment, that space is conserved in noncritical system services.

A more direct comparison can be made between our algorithms and the commercially available packages by replacing the relevant code with our input and output routines, but leaving the extra functionality of the commercial routines untouched. We chose the GNU routines for this experiment because the code is available (the UNIX code is proprietary) and the GNU routines outperform those in the other two packages. We replaced only the date translation code in the GNU routines with the full versions of the input and output algorithms, making our routines comparable to the GNU definition of input and output. Since the GNU routines use a 32-bit timestamp, we changed both the code and tables for the full input and output algorithms to use only 32-bit instructions. We then tested the GNU `mktime()` against the modified `mktime()` (incorporating the full input algorithm) and the GNU `gmtime()` against the modified `gmtime()` (incorporating the full output algorithm). Figures 18 and 19 show the results for input and output, respectively. The figures show that the modified algorithms outperform their GNU counterparts considering time efficiency. However, the GNU routines are smaller; `gmtime()` is 792 bytes in size compared to 22K bytes for the modified `gmtime()` while `mktime()` is 4K bytes in size compared to 26K for the modified routine. If this space penalty is too severe, the slender algorithm could be used instead of the full algorithm, reducing the space costs by at least a factor of two, while retaining constant time performance.

Figure 17: A plot of output speed over the range of a UNIX timestamp

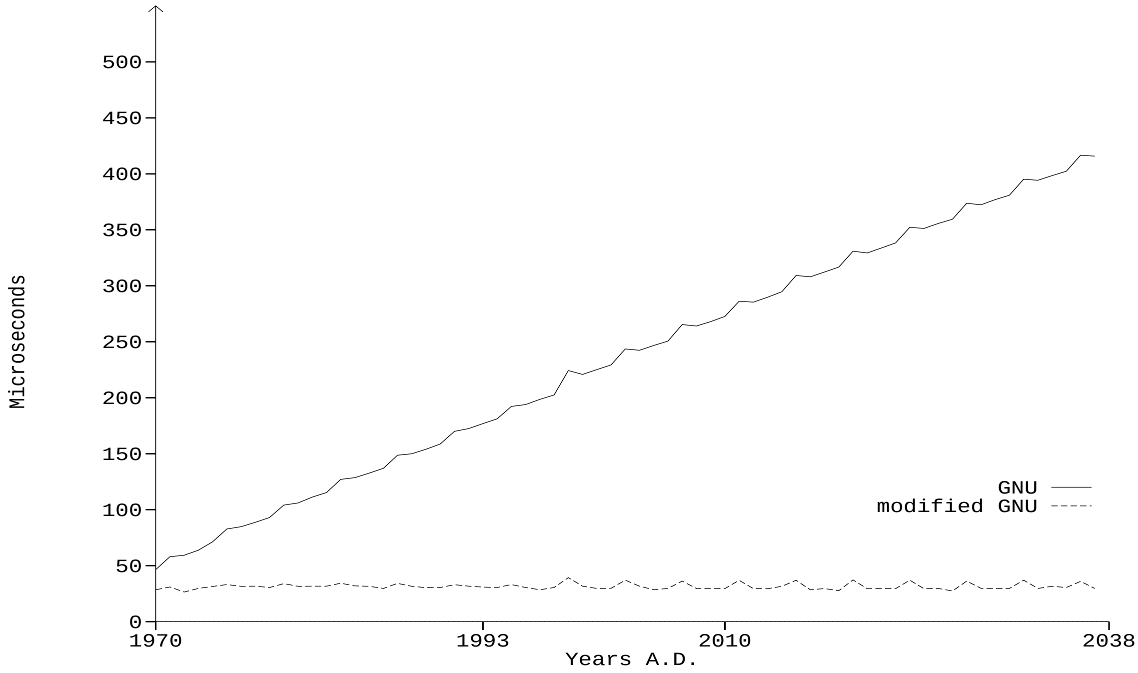

Figure 18: A direct input comparison

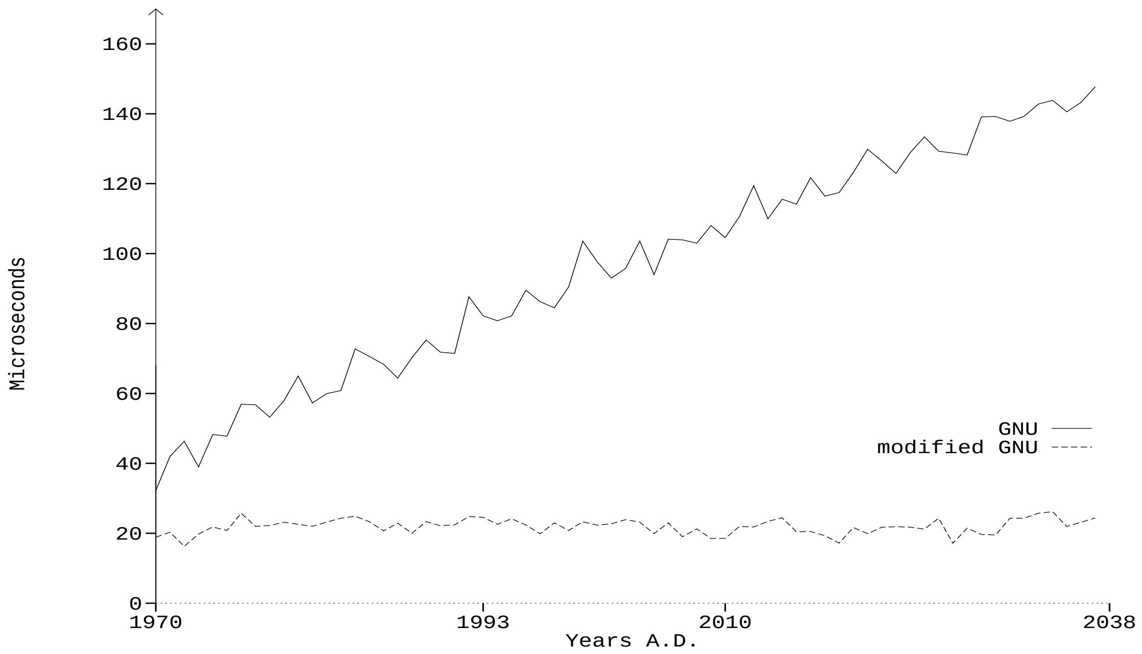

Figure 19: A direct output comparison

7 Non-Gregorian Input and Output

In general, the techniques developed in the previous sections for efficient input and output in the Gregorian Calendar apply to input and output in other calendar systems. Calendars typically relate a count of short-term celestial cycles, such as mean solar days (revolutions of the Earth about its axis), to long-term celestial cycles, such as years (orbits of the Earth about the Sun). For example, the Gregorian calendar counts tropical years (the year with respect to the seasons) in terms of days while the traditional Chinese calendar counts lunar months (from new moon to new moon) and 19-year periods known in the West as Metonic cycles [Fraser 1987], in terms of days. Since there are not (currently) an integral number of days in either a lunar month, a tropical year, or a Metonic cycle, rule-based adjustments, called *intercalations*, are made to keep the count accurate. In the Gregorian calendar, the intercalary rules add a leap day in some years. In the Chinese calendar, a leap month is added to certain years and month lengths are adjusted so that the new moon occurs on the first of every month (there are currently $29\frac{1}{2}$ days in a lunar month). The intercalations are what make input and output difficult.

The key to applying the techniques discussed in previous chapters is to identify a long-term fixed period in a calendar and reduce the intercalations to table-lookups. The particular fixed period depends on the calendar. For the Gregorian calendar, we used the greg, a 146097 day period, but other periods exist for different calendars. For instance, the Julian calendar, which adds a leap day every four years, has a fixed period of four years ($365 \times 3 + 366$ days) while the Chinese calendar has a 60-year cycle [Fraser 1987].

8 Summary

Temporal databases require timestamps with much greater range than that available in the 32-bit integer representation commonly used in operating systems. Previously we proposed using a 60-bit signed integer representation, which covers all past time. However, this representation renders existing input and output algorithms, which exhibit linear complexity, unacceptable inefficient.

We presented a series of algorithms for two tasks, input and output of Gregorian dates. The most important optimization was to identify a *constant cycle* for which there are a fixed number of all relevant intervals. The 400-year *greg* for the Gregorian calendar is used for this purpose (input and output in the Chinese calendar might use a 60-year cycle). The duration of the constant cycle is effectively the greatest common multiple of the duration of the underlying variable cycles. For example, while the length of months and years vary in the Gregorian calendar, each greg has the same number of days, months, and years. Using constant cycles reduces the complexity from linear in the input time to constant. Fortunately, calendars, which exist to relate short term cycles such as hours or days to longer term cycles such as seasons, generally incorporate constant cycles in their definition, if only to make calculations, which until recently were done by hand, tractable. It is satisfying to see how such cycles can dramatically speed up computer algorithms that make similar calculations.

We applied the following additional techniques in designing these algorithms.

- Make allowance for leap seconds, which are counter to the notion of a constant cycle, *after* doing the input conversion and *before* doing the output conversion.
- Cache the results of functions with small domains or ranges. For example, the number of leap days preceding each year in a greg can be stored in a 400-byte table. Another example is a division and modulus by a constant (365) of a value in the range 0–149067, where a binary search tree of 400 leaves suffices.
- Instead of doing a brute-force calculation, make an initial, optimistic guess and then correct it. The correction will often require less work. An example is determining the year within a greg.
- Perform a series of expensive multiplication, division, and modulus operations on 59-bit timestamps, as well as 36-bit year values, in parallel, accumulating the result by examining the value bit-by-bit (actually, in groups of 4 bits).

The cumulative effect of these optimizations reduced the time for Gregorian input to about 8 μsec and Gregorian output down to about 22 μsec. For comparison, the best available algorithms, found in the GNU C library, would take about 11.5 msec and 3 msec, respectively, to output a date in 1993, assuming an origin of 1 A.D. As several of the optimizations traded space for time, our implementations were bigger by a factor of 10.

Given the efficiency of the final algorithms, it is doubtful that a further substantial decreases in execution time are possible. Focusing on reducing the size of the tables will perhaps be more profitable.

Acknowledgements

This work was supported in part by NSF grant ISI-8902707 and IBM contract #1124.

References

[Allen 1981] Allen, J.F. "An Interval-Based Representation of Temporal Knowledge," in *Proceedings of the International Joint Conference on Artificial Intelligence*. Vancouver: 1981, pp. 221–226.

[Ariav 1986] Ariav, G. "A Temporally Oriented Data Model." *ACM Transactions on Database Systems*, 11, No. 4, Dec. 1986, pp. 499–527.

[Aveni 1989] Aveni, A. F. "Empires of Time: Calendars, Clocks, and Cultures." New York: Basic Books, Inc., 1989.

[Clifford & Tansel 1985] Clifford, J. and A.U. Tansel. "On an Algebra for Historical Relational Databases: Two Views," in *Proceedings of ACM SIGMOD International Conference on Management of Data*. Ed. S. Navathe. Association for Computing Machinery. Austin, TX: May 1985, pp. 247–265.

[Clifford & Rao 1987] Clifford, J. and A. Rao. "A Simple, General Structure for Temporal Domains," in *Proceedings of the Conference on Temporal Aspects in Information Systems*. AFCET. France: May 1987, pp. 23–30.

[Dyreson & Snodgrass 1993] Dyreson, C. E. and R. T. Snodgrass. "Timestamp Semantics and Representation." *Information Systems*, 18, No. 3, Sep. 1993.

[Fraser 1987] Fraser, J. "Time the Familiar Stranger." Redmond, WA: Tempus Books, 1987.

[Hawking 1988] Hawking, S. "A Brief History of Time." New York: Bantam Books, 1988.

[Liskov & Guttag 1986] Liskov, B. and J. Guttag. "Abstraction and Specification in Program Development." New York: McGraw-Hill, 1986.

[Soo & Snodgrass 1992] Soo, M. and R. Snodgrass. "Mixed Calendar Query Language Support for Temporal Constants." TempIS Technical Report 29. Computer Science Department, University of Arizona. Revised May 1992.

[Soo et al. 1992] Soo, M., R. Snodgrass, C. Dyreson, C. S. Jensen and N. Kline. "Architectural Extensions to Support Multiple Calendars." TempIS Technical Report 32. Computer Science Department, University of Arizona. Revised May 1992.

[Tansel, A. et al. 1993] Tansel, A., J. Clifford, S. Gadia, S. Jajodia, A. Segev and R. Snodgrass (eds.). "Temporal Databases: Theory, Design, and Implementation." Database Systems and Applications Series. Redwood City, CA: Benjamin/Cummings, 1993.