# Trading Space for Time: Constant-Speed Algorithms for Managing Future Events in Scientific Simulations

**Clarence Lehman**[1], **Adrienne Keen**[2], **and Richard Barnes**[1]
[1]University of Minnesota, 123 Snyder Hall, Saint Paul, MN 55108, USA
[2]London School of Hygiene and Tropical Medicine, Keppel St., London WC1E 7HT, UK

*"It is a mistake to try to look too far ahead. The chain of destiny can only be grasped one link at a time."*
—Winston Churchill

**Abstract**— *Given vast increases in computing capacity, applications in science and engineering that were formerly interpreted with ordinary or partial differential equations, or by integro-partial differential equations, can now be understood through microscale modeling. Interactions among individual particles—be they molecules, viruses, or individual humans—are modeled directly, rather than first abstracting the interactions into mathematical equations and then simulating the equations. One approach to microscale modeling involves scheduling all events into the future, wherever that is possible. With sufficient space-for-time tradeoffs, this considerably improves the speed of the simulation, but requires scheduling algorithms of high efficiency. In this paper we describe our variation on calendar queues and their usage, presenting detailed algorithms, intuitive explanations of the methods, and notes from our experiences applying them in large-scale simulations. Results can be useful to scientists in ecology, epidemiology, economics, and other disciplines that employ microscale modeling.*

**Keywords:** microscale modeling, discrete event simulation, calendar queues, pending events set, space-time tradeoff

## 1. Introduction

The obvious approach to model a large number of discrete interacting entities, hereinafter called "individuals," is to emulate what is done to model continuous systems with differential equations. That is, select a small time step $\Delta t$, compute how the system will change during the interval $\Delta t$, update the system with those changes, then advance to the next time step. In ordinary differential equations, as the time step shrinks, the dynamics of the simulated system converge to the correct behavior. This is "macroscale modeling," following Euler's method or its many variations [1]. With a model of 100 compartments, representing, for example, 100 age classes in a human population, relatively few dynamical variables must be examined and updated in each time step.

The same approach works with microscale modeling, though with difficulties. At each time step, each individual is examined to determine what interactions will occur during that time step. The difficulty with this approach is twofold. First, each individual acts as a dynamical variable, so there can be many millions or hundreds of millions of variables to be examined and updated in each time step. Moreover, as the time step shrinks to assure convergence, it becomes exceedingly unlikely that anything will happen to a given individual during the time step. Therefore, in contrast with its macroscale counterpart, that approach to microscale modeling spends most of its time checking and finding nothing to do.

Inspiration for a faster approach comes from an alternative method of solving differential equations. Instead of determining what will happen during the present small time step, an algorithm can determine at what time in the future the next event will occur. This can be determined reliably for the very next event, and the precise process for doing so is called Gillespie's method [2]. It is the complement of the standard method.[1]

Despite certain epistemological difficulties about projecting the future that are beyond the scope of the present paper, hinted at in Churchill's statement above, Gillespie's method can be extended to determine possible times for all future events in many dynamical systems of scientific interest—or at least all events that control the fate of the system. But the number of future events can be large, with many events per individual, and the number of individuals in the simulation may be tens or hundreds of millions or more.

Fortunately, algorithms are known that are extremely efficient at handling schedules of future events. Discovered by Randy Brown in 1988 [3], these are called "calendar queues" or "pending event sets," and have been undergoing successive refinements ever since (e.g. [4] [5] [6] [7]). They have the desirable—and remarkable—property that their speed is independent of the number of events scheduled.

[1]In simulating $f(x) = dx/dt \approx \Delta x/\Delta t$, a small time step $\Delta t$ can be established, such as 0.01 seconds, and the change in population (or other simulated quantity) can be estimated as $\Delta x \approx f(x)\Delta t$. That is Euler's method. Alternatively, a change in quantity $\Delta x$ can be specified (such as a population growth of one individual) and the time for that to occur can be estimated as $\Delta t \approx \Delta x/f(x)$. That is Gillespie's method. Thus the mathematics for the two are complementary.

Adding an event, canceling one, or finding the next event about to occur is the same whether the schedule contains 100 events or 100 million. That is, they are "Order 1" algorithms.

In this paper we present our adaptation of calendar queues to large-scale individual-based modeling in epidemiology. Lessons should be applicable to areas including ecology, economics, and other physical and natural sciences. We attempt to make our presentation intuitive for access by scientists and other readers outside computer science. The goals of this paper are to (1) review the idea of space-for-time trade-offs that have become widely useable and applicable to other algorithms (e.g. [8]), (2) explain our variation on calendar queues and their incorporation in microscale simulations, and (3) present our algorithms in full detail for use and adaptation by others.

## 2. Space for time

The persistent increase in random access computer memories has carried algorithms through a "phase change," wherein a slow continuous advance in memory sizes has resulted in a rapid, almost abrupt, change in some of the rules for constructing algorithms for scientific programs. If it will speed processing, computer algorithms can now afford to allocate hundreds of millions of bytes of empty space—even if that space will never be used. This is a "space-for-time tradeoff." With large memories now available, such allocation is no longer wasting memory. On the contrary, leaving memory unused, or leaving it applied to insignificant purposes, is wasting it.

A basic space–time tradeoff arises with numerical keys. Suppose we have 10,000 items, each identified by a distinct six-digit "key," and with keys randomly distributed among values from '000000' to '999999'. Suppose each of the 10,000 items occupies 100 memory cells (e.g. 100 bytes). Stored contiguously, this will require $10^4 \times 10^2 = 10^6$ memory cells. In such a compact arrangement, searching can be relatively fast if the entries are kept in numeric order[2]. However, in this case adding and deleting will be slow, averaging $N$ or more accesses to keep the list contiguous and in order. On the other hand, if the entries are left in random order, adding and deleting will be fast, 1 to 3 accesses only,[3] but searching will be slow, sequentially checking each entry until the right one is encountered. The point is, this minimal-space approach inevitably results in algorithms that are slow in one respect or another.

An alternative is to "waste" memory by allocating one slot in memory for each of the million entries possible. Now to search for a specific six-digit key, say key '314159', the algorithm merely goes directly to the 314,159th entry of the table. Only one access to the memory array is thus needed to retrieve, and the same is needed to add or delete. With 10,000 active entries, this space–time tradeoff speeds the algorithm 5,000 fold. However, it comes at the expense of 100-million memory cells, about one-tenth of a gigabyte. Such cavalier abandon in the use of memory would have been unthinkable until recently, but if speed is the utmost criterion, then allocating an extra 1/10 GB to accomplish a multi-thousand-fold increase in speed is the clear and proper choice.

This approach extends to larger keys through the method of "hash coding," which is directly related to calendar queues. Hash coding is an Order-1 algorithm known at least since Arnold Dumey in 1956 [9]. The key may be an individual's first, last, and middle name, for which the space required for direct access would be astronomical, beyond the power of any computer presently foreseeable. Even if the key was only a nine-digit social security number, such as 123-45-6789, providing one direct-access entry for all possible social security numbers would be prohibitively large.

The simplest solution merely extracts the rightmost six digits of the social security number and indexes an array of a million entries with those six digits. Of course, as many as 1,000 individuals may share the same last six digits of their social security numbers, so "collisions" can occur. But with only 10,000 entries of a million active, and assuming all possible social security numbers are equally likely, each entry in the array has only a 0.01 chance of being occupied, so the chance that two or more individuals will occupy the same cell is very small. Nonetheless, the possibility of collisions must be provided for, and a variety of practical methods have been devised [10]. Once that is done, locating an individual by social security number, or indeed by first, last, and middle name, can be accomplished in one access, or arbitrarily close to one access, with a sufficiently large space-for-time tradeoff.

Dumey's scheme [9] was to use a modulus operation by considering the key to be a large number, dividing it by the size of the memory array (number of entries in the array), then discarding the quotient and using the remainder to index the array—as in the social security example. In that case, the rightmost six digits were equivalent to the remainder after division by one million. Essentially the same underlying scheme is applied in calendar queues, dividing the scheduled time by the size of the memory array (one year's worth of minutes in the intuitive example to follow), and using the remainder to index the array. Therefore, the same space-for-time tradeoffs that make hash-coded accesses maximally fast also can make calendar queues, properly programmed, maximally fast for managing large numbers of future events.

## 3. Future events

Having emphasized the value of spending memory to buy time, we must also say that it is pointless to spend memory

[2]For instance, by using a binary search algorithm, which is of Order $\log_2 N$ accesses, where $N$ is the number of items in the list

[3]New entries can be added at the end in 1 access; deleted entries can be swapped with the entry at the end in 3 accesses.

when it does not buy time. The more events that are scheduled at once, the greater the amount of memory that is needed to handle them efficiently, in direct proportion to the number scheduled. Also, the more that the number of events scheduled vary during the simulation, the more frequently the data structures should be optimized by "resizing" [3].

Therefore, to help keep the scheduling algorithms efficient, our microscale simulation programs withhold all but one event per individual from them. Characteristics of individuals are maintained in a large array of data structures, $A[n]$, indexed by individual number $n$, which ranges from one to some maximum value. This array includes data of two types: (1) information about the individual, such as, in a model of human events, date of birth, sex, geographic location, and so forth, and (2) a list of all future events relevant to that individual. This large array is not processed nor examined by the scheduling routines described in this paper.

Only the earliest among the events pending for each individual is entered into the global schedule, with the data structure $A[n]$ holding the rest. Such withholding of information has several benefits: (1) the number of events managed by these algorithms is considerably reduced, (2) the number of events that must be canceled and rescheduled is reduced, and (3) the size of the scheduling data structures are predictable, with precisely one event per individual. This partly obviates the need for the scheduling algorithms to maintain separate lists for near, intermediate, and far future events, as in some variations of calendar queues [11], and also eliminates the need for time-consuming "resizing" operations [3].

## 4. Intuitive view

We want to (1) schedule new events, (2) cancel existing events, and (3) notify a dispatcher as the time for each event arrives—all three with maximal efficiency. The coding details can be subtle, but the overall operation is not. It can be understood intuitively through a physical analogy.

Assume, for a specific illustration, that half a million events are to be scheduled over the next five years, and that they appear more-or-less randomly throughout that period. Suppose that each event has a ticket with (1) a unique event number and (2) a scheduled time, represented at least to the nearest second, but possibly much finer.

Now consider a series of pigeon-hole bins to contain the tickets, one bin representing each minute of an entire year. The first bin represents the first minute after midnight on New Year's Day, the second bin represents the second minute, and so forth to the last bin, which represents the last minute on December 31st. That is 366 days $\times$ 24 hours/day $\times$ 60 minutes/hour = 527,040 bins total, each labeled with the month, day, hour, and minute that it represents. Each bin also has a flag that can be lowered or raised according to whether the tickets in the bin are known to be in chronological order. We assumed half a million events to be scheduled, less than one event per bin on average.

### 4.1 Creating a new event

Events are created as the simulation proceeds, each associated with a particular individual and with a precisely assigned time, usually stochastically assigned. In an ecological model these may represent a time of birth or death, in an epidemiological model they may represent the time of onset of a disease, or the time for transmission to another individual. In any case, new events arise frequently during the simulation. The procedure for scheduling a new event is quite easy:

1. Go to the bin representing the month, day, hour, and minute for the event. Although the year, second, and any fraction of a second are not used to select a bin, they are later used to place events in precise chronological order.
2. Drop the event's ticket on top of the others in the bin.
3. Raise the flag on the bin to indicate that its tickets may no longer be in chronological order.

That required only a single operation, regardless of how many events were in the bin. We take it to be important merely to drop the ticket atop others in the bin, as above, rather than trying to sort it into place among other tickets in the bin. Earlier implementations of calendar queues [3] keep all bins always sorted, but that can be disabling if a large number of events accumulate in any bin. Such accumulation can occur during testing or simulation.

### 4.2 Canceling an existing event

Once scheduled, events may occasionally have to be canceled. For example, in an epidemiological model, a healthy individual may become the target of an infection. Whatever the next event in their life was, it may have to be rescheduled as the simulated individual progresses toward disease and infectiousness. Therefore, the existing event will be canceled and the earliest of other future events for the individual will be scheduled instead. In the physical analogy, that requires three steps:

1. Go to the bin representing the month, day, hour, and minute for the event. As before, ignore the year, second and any fraction of a second.
2. Flip through them to find the ticket for the event in question.
3. Destroy that ticket.

That required one operation for every ticket in the bin, but on average there is only one ticket in the bin. Canceling an event can be slow if the events cluster badly, because of the need to flip through the tickets in the bin. But canceling is not a usual operation. The two common operations are adding events (described above) and dispatching them (described next).

### 4.3 Dispatching the next event

The simulation proceeds stepwise by locating the earliest among all events in the schedule, removing it, then processing it. This is efficient, but it involves several steps:

1. Go to the bin representing the current day, hour, and minute.
2. If the flag on the bin is raised, arrange the tickets in chronological order and lower the flag.
3. Leave any tickets for future years in the bin.
4. Process any tickets from this year, day, hour, and minute, each to be handled precisely in sequence as the scheduled second and fraction of a second arrives.
5. If any tickets for the current bin arrive while the bin is being handled, put them in their proper position among the other tickets.

This required only one operation for each ticket, plus one or two more per ticket to order them chronologically before dispatching the contents of the bin. Again, on average there is only one ticket in the bin.

This method intentionally does not keep tickets in the bins ordered, using instead "just-in-time sorting." Usually this will make little difference, since the bins are intentionally designed to be nearly empty. However, as described earlier, if unexpected clustering occurs, this just-in-time sorting will be much faster than keeping the contents of all bins in order each time an event is added.

Within a simulation program using these scheduling algorithms, the individual associated with the ticket being handled will have other pending events in its entry of data structure $A[n]$. The simulation program will then pass the earliest of these to the scheduling algorithms, through a call to *EventSchedule*.

The discussion above shows how the algorithms achieve their speed—by maintaining at least as many bins as there are tickets. If there were sixty times as many tickets—thirty million—the same speed of operation could be maintained simply by increasing the number of bins by sixty, to one bin for each second.

## 5. Applications

The algorithms described here have been applied and tested in a large-scale multi-compartmental epidemiological model of tuberculosis transmission developed by one of us (A.K.). That model runs with upwards of $6 \times 10^7$ individuals (60 million), representing the entire population of the UK, on multiple parallel processors for parameter fitting by simulated annealing. Each individual has many events pending, including, for example, scheduled times of death, emigration, onset of disease for recently infected individuals, next transmission for infectious individuals, potential vaccination for juveniles, and so forth.

In this epidemiological model, typical runs spanned 30 simulated years and used 75 million bins occupied by 60 million individuals. Each run consumed about 80 seconds on a 2.8 GHz processor, using a little over 6 GB of memory on each of 30 to 50 parallel processors. The average time increment between scheduled events was 14 simulated seconds, with a standard deviation of 12 seconds. The minimum was less than a simulated microsecond, whenever stochastic events appeared by chance close together in time. The maximum time increment was 53 simulated seconds. Thus the time steps are very small compared with a corresponding macroscale model.

In simplified timing tests on the same processor, outside of the operation of the epidemiological model, a list with $6 \times 10^7$ individuals needed 30 nanoseconds on average to schedule each new event, 18 nanoseconds to cancel an event, and 12 nanoseconds to dispatch each event when its time arrived. This was near-ideal conditions, with new events arising in sequence in a way that minimized clustering in the schedule. Expanding the number of individuals by a factor of more than 16, to $10^9$ individuals (one billion) required exactly the same amount of time per operation—within small bounds of statistical error—demonstrating the Order-1 behavior of the algorithms.

On the other hand, events arising in random order needed 90 nanoseconds to schedule each new event into a list of 60 million and 180 nanoseconds into a list of one billion. The three to six-fold increase can be attributed to interactions with internal memory caches. Such caches grow less useful as memory accesses become less localized.

## 6. Algorithmic details

The intuitive picture sketched above converts directly into the algorithms displayed in the appendix. As implemented in the algorithms, the bins need not correspond to standard time units such as minutes, but can be any values.

A simulation begins by adding one or more events, typically one event per individual, and ends either at a predetermined time or when the last event has been dispatched. Array $A[n]$ would be established earlier with a collection of pending events for each individual $n$. The main simulation program would be structured as follows:

```
[1] ProgramInit();
[2] loop for all $n$ in $A[n]$ :
        $EventSchedule(n,\ earliest(n))$;
[3] loop for $t$ from $0$ to $t_{max}$:
        $Process(EventNext())$;
[4] exit;
```

In step 1 above, *ProgramInit* sets the initial conditions for the program, including allocating all individuals that will start the simulation and all future events that are known for each. Step 2 moves through all individuals, selects the earliest event for each ($earliest(n)$), and schedules each event by calling *EventSchedule*. With all events to start the simulation scheduled, step 3 repeatedly asks for the next chronological event by calling *EventNext* and passing the number of that event to *Process*. In turn, *Process* will call upon *EventSchedule* and possibly *EventCancel* and *EventRenumber* while carrying out the simulation. *ProgramInit*, *Process*, and *earliest*, as well as array $A[n]$, are written as part of the simulation program. The rest are scheduler algorithms detailed in the appendix.

The two main data structures organizing the earliest event for each individual are (1) a circular array of integers $Q[h]$, each heading a linked list in $P[n]$ of events scheduled for time bin $h$, and (2) an array of integers $P[n]$, each continuing the linked list from $Q[n]$. The number of entries in $P[n]$ must equal the number in external array $A[n]$, and like $A[n]$, $P[n]$ is indexed by individual number. But the number of time bins $Q[h]$ may be smaller or larger than the number of individuals. The size of $Q[n]$ is a matter of optimization. It is typical to have one time bin for each event that could be scheduled, meaning each bin will represent a single event on average. A space–time tradeoff occurs because optimal allocation leaves about one-third of the bins empty.[4]

Each bin $Q[h]$ represents many related times, all equal modulo the width of the series of time bins, $Qw$. The width $Qw$ of all bins combined is also a matter of optimization. If it is much too large, events will tend to cluster near the bin being dispatched. If it is much too small, events will tend to spread out, with most bins containing events that are for the more distant future. A suitable value for $Qw$ can be found by knowledge of the system being analysed, or by experimental trials to find a good speed of operation.

For speed of addition, the lists of events in $P[i]$ are not maintained in any particular order, but each bin is sorted chronologically before it is dispatched. Any sorting algorithm used should have (1) best performance when the list is already partially sorted, e.g. Order $N$, important because lists will remain partially sorted from earlier passes, (2) high-speed when sorting only 1 and 2 entries, which are the most common, and (3) good worst-case performance, e.g. Order $N \log_2 N$. The sorting routine presented as Algorithm 5 in the appendix has these properties.

## 7. Conclusions

The algorithms presented here can be incorporated into any individual-based or other microscale model, where they can speed simulations many orders of magnitude over alternative methods that are not Order-1.

They are part of a large-scale simulation model developed by one of us (A.K.) for tuberculosis in the UK. Sixty million individuals thus can be handled by allocating less than a gigabyte of random access memory—within the reach even of portable computers. In practice, these algorithms should be able to schedule, cancel, and dispatch up to $10^7$ or more events per second with 60 million or more pending events maintained in the queue. Therefore, they should not become a bottleneck in the simulation as a whole.

Compilable copies of the code described here and related simulation algorithms are available free from the authors upon request.

[4]Under random distribution, $1/e = 37\%$ will be empty. That can be shown to be optimal for overall speed if all bin operations are equally fast.

## 8. Acknowledgements

We thank Fred Lehman, Holly MacCormick, Peter Hawthhorne, Ben Kerr, Celia Hemmerich, and Shelby Williams for discussions and encouragement. The project was supported in part by a resident fellowship grant to C.L. from the UMN Institute on the Environment, by grants of computer time from the Minnesota Supercomputer Institute, and by doctoral research funding to A.K. from the Modelling and Economics Unit at the Health Protection Agency, London.

## 9. Contributions

C.L. considered the notion of stochastically prescheduling the earliest future event for each individual and developed an initial application. A.K. expanded the approach for her large-scale tuberculosis model, leading to refinements in the algorithms. A.K. conceived a grouping method to work in concert and make these algorithms practical for multi-compartment simulation models [8]. R.B. participated in the evaluation, applications to other areas, and the literature review. C.L. coded the algorithms and A.K. tested them in large-scale operation. All authors contributed to the manuscript.

## References

[1] W. H. Press, S. A. Teukolsky, W. T. Vetterling, and B. P. Flannery, "Numerical recipes: The art of scientific computing, third edition," *Cambridge University Press, New York*, 2007.

[2] D. T. Gillespie, "Exact stochastic simulation of coupled chemical reactions," *Journal of Physical Chemistry*, vol. 81, pp. 2340–2361, 1977.

[3] R. Brown, "Calendar queues: a fast 0(1) priority queue implementation for the simulation event set problem," *Commun. ACM*, vol. 31, no. 10, pp. 1220–1227, Oct. 1988.

[4] R. Rönngren, J. Riboe, and R. Ayani, "Lazy queue: an efficient implementation of the pending-event set," *SIGSIM Simul. Dig.*, vol. 21, no. 3, pp. 194–204, Apr. 1991.

[5] G. A. Davidson, "Calendar p's and q's," *Communications of the ACM*, vol. 32, pp. 1241–1242, 1989.

[6] T. Hui and I. Thng, "Felt: A far future event list structure optimized for calendar queues," *Simulation*, vol. 78, no. 6, pp. 343–361, 2002.

[7] G. Yan and S. Eidenbenz, "Sluggish calendar queues for network simulation," in *Modeling, Analysis, and Simulation of Computer and Telecommunication Systems, 2006. MASCOTS 2006. 14th IEEE International Symposium on*, 2006, pp. 127–136.

[8] A. Keen and C. Lehman, "Trading space for time: Constant-speed algorithms for grouping objects in scientific simulations," *Proceedings, International Conference on Scientific Computing.*, pp. 146–151, 2012.

[9] A. I. Dumey, "Indexing for rapid random access memory systems," *Computers and Automation*, vol. 6, pp. 6–9, 1956.

[10] D. E. Knuth, "The art of computer programming, volume 3: Sorting and searching, second edition," *Addison-Wesley, Reading, MA*, 1998.

[11] R. Goh and I. Thng, "Mlist: An efficient pending event set structure for discrete event simulation," *International Journal of Simulation-Systems, Science & Technology*, vol. 4, no. 5-6, pp. 66–77, 2003.

## 10. Appendix

To use the algorithms described in this paper, it is only necessary to understand the entry and exit conditions that appear at the beginning of each, not the code itself. Nonetheless, to allow complete evaluation of the algorithms, and to encourage further development of them, we present them as pseudo-code inspired by and simplified from the programming languages C, Python, and R. The algorithms are defined with sufficient precision that they can be run, tested, timed, modified, or translated to other languages. Familiarity with a relatively few operators* and with the syntax of flow control (if, for, while, etc.), is sufficient to follow the algorithms. *WarnMsg* and *ExitMsg* display error messages and the latter terminates the program. Not all functions return values. Text copies of this pseudo-code translated into operational C are available from the authors upon request, or from the associated website `www.cbs.umn.edu/modeling`.

**PROGRAM PARAMETERS**

| | |
|---|---|
| $TN \equiv (100000000)$ | Example, maximum number of time bins. |
| $PN \equiv (100000003)$ | Example, maximum number of forward indexes to time bins. |
| $TW \equiv 20$ | Example, time width of all bins combined (for optimization). |

**INTERNAL DATA STRUCTURES**

| | | | |
|---|---|---|---|
| $PZ \equiv -1$ | | | Marker for empty bins. |
| **real** | $T[PN]$ | $\leftarrow 0;$ | Time for each scheduled event. |
| **integer** | $P[PN]$ | $\leftarrow PZ;$ | Forward indexes within bins, ending with zero. |
| **integer** | $Q[TN]$ | $\leftarrow 0;$ | First index for the bin, with zero for empty bins, negative for unsorted bins. |
| **real** | $Qw$ | $\leftarrow TW;$ | Interval of time represented for each cycle in $Q$. |
| **integer** | $Qn$ | $\leftarrow TN;$ | Number of elements in $Q$. |
| **integer** | $Qi$ | $\leftarrow 0;$ | Index of the immediate time bin. |
| **integer** | $Qe$ | $\leftarrow 0;$ | Number of events in all bins. |
| **real** | $Qt0$ | $\leftarrow 0;$ | Earliest time representable this cycle in $Q$. |
| **real** | $Qt1$ | $\leftarrow TW;$ | Earliest time beyond this cycle in $Q$. |
| **real** | $t$ | $\leftarrow 0;$ | Current time, last dispatched event. |

**Algorithm 1. SCHEDULE A NEW EVENT**

**Upon entry to the algorithm, (1)** $n$ contains the number (starting with 1) of a new event. **(2)** $te$ contains the time at which the new event will occur. **(3)** $P[n]$ indicates that the event is unscheduled (equal to $PZ$). **(4)** The scheduling data structures are prepared as described above. **At exit, (1)** the event has been scheduled, to occur when the proper time arrives. **(2)** $T[n]$ records the time $te$ of the event. **(3)** $P[n]$ links the event with others in its time bin.

```
EventSchedule(n, te)  integer n, real te; integer i; real tr;
  if n < 1 or n ≥ PN: ExitMsg(3);            1. Check the index and make sure an
  if P[n] ≠ PZ: ExitMsg(4);                      event is not already scheduled
  if te < t: ExitMsg(5);                          and is not in the past.

  te → T[n];                                   2. Record the time of the new event.

  (te − Qt0)/Qw → tr; tr − (int)tr → tr;       3. Convert the time to a bin number.
  tr∗Qn → i;

  abs(Q[i]) → P[n], −n → Q[i], ↑Qe;            4. Add the event to the list for that bin
                                                  and increment the number of events.
```

* *The pseudo-code given here is two-dimensional, as in the language Python, so that indentation completely defines the nested structure, with no need for bracketing characters such as '{' and '}'. Variables and function names are italicized and flow control and reserved words are bolded.*

*The assignment operator is represented either as '←' or '→', similar to assignments in R. The compound assignments '$a + 1 \to a \to b \to W[i][j]$' and '$W[i][j] \leftarrow b \leftarrow a \leftarrow a + 1$' are equivalent, first incrementing $a$ and placing the results back in $a$, then in $b$, and then in the $i, j$th element of the array $W$.*

*The expression structure '$c\,?\,u : v$', where $c$ is a condition, $u$ is an if-expression, and $v$ is an else-expression, follows that of C. Using up-tick and down-tick operators to write '↑a', '↓a', 'a↑', and 'a↓' form pre- and post-increments by one, as in '`++a`', '`--a`', '`a++`', and '`a--`' of C.*

*Arrays are indexed as in the language C, starting with 0. Data types are '**integer**' and '**real**', with the latter specifying floating point. Operator precedence is that of C, with assignments having lowest precedence. Logical operators such as '**and**' and '**or**' are preemptive, terminating a chain of logical operations as soon as the result is known. Permanent global assignments, as would be represented '`#define` $\alpha\ \beta$' in C, are rendered as '$\alpha \equiv \beta$'.*

**Algorithm 2. CANCEL AN EXISTING EVENT**

**Upon entry to the algorithm, (1)** $n$ contains the number (starting with 1) of the event to be cancelled. **(2)** $T[n]$ contains the scheduled time of the event. **(3)** the scheduling data structures are prepared as described above. **At exit,** the event has been removed from the list.

```
EventCancel($n$)   integer $n$; integer $i$, $j$, $jp$; real $tr$;
  if $n < 1$ or $n \geq PN$:  $ExitMsg(6)$;                   1. Check the index and make sure an
  if $P[n] = PZ$:  $ExitMsg(7)$;                                 event is scheduled.

  $(T[n] - Qt0)/Qw \ \rightarrow tr$;  $tr - (int)tr \ \rightarrow tr$;        2. Convert the time to a bin number,
  $tr * Qn \ \rightarrow i$;                                        modulo the duration of the cycle.

  if $subcancel(n,\ i)$:  return;                              3. Remove it from its normal bin
  $(i - 1 + Qn)$ mod $Qn \ \rightarrow i$;  if $subcancel(n,\ i)$:  return;      or from an adjacent bin above or
  $(i + 2 + Qn)$ mod $Qn \ \rightarrow i$;  if $subcancel(n,\ i)$:  return;      below (due to rounding error).

  $ExitMsg(8)$;                                             4. If the specified event was not in
                                                               the list, signal an error.

integer $subcancel(n,\ i)$   integer $n$, $i$; integer $j$, $jp$;
  $0 \ \rightarrow jp$,  $abs(Q[i]) \ \rightarrow j$;
  loop while $j > 0$:                                       1. Scan the list of pending events in
    if $j = n$:                                                this bin and remove the specified
      if $jp > 0$:  $P[j] \ \rightarrow P[jp]$;                          event. (The average number of events
      else $Q[i] > 0 ? P[j] : \ - P[j] \ \rightarrow Q[i]$;                  in non-empty bins is about 1.5)
      $PZ \ \rightarrow P[j]$;  if $\downarrow Qe < 0$:  $ExitMsg(9)$;
      return 1;

    $j \ \rightarrow jp$,  $P[j] \ \rightarrow j$;

  return 0;
```

**Algorithm 3. DISPATCH THE NEXT EVENT**

**Upon entry to the algorithm, (1)** $T$ contains the time for each scheduled event. **(2)** The scheduling data structures are prepared as described above. **At exit, (1)** $EventNext$ contains the number of the next event. If zero, no events are scheduled. **(2)** $t$ contains the time of the next event, if $NextEvent$ is not zero.

```
integer $EventNext()$   integer $j$, $n$;
  loop while $Qe > 0$:
    loop while $Qi < Qn$:                                   1. Advance to the next non-empty
      $Q[Qi] \ \rightarrow j$;  if $j = 0$:  $\uparrow Qi$;  repeat loop;              bin.

      if $j < 0$:                                           2. Sort the bin if it may be necessary
        $sort(P,\ -j,\ 0,\ order) \ \rightarrow Q[Qi] \ \rightarrow j$;                (usually sorts 1 or 2).

      if $T[j] < Qt1$:                                      3. If the event belongs to this pass,
        if $P[j] = PZ$:  $ExitMsg(2)$;                           remove it, decrement the number of
        $P[j] \ \rightarrow Q[Qi]$,  $PZ \ \rightarrow P[j]$,  $\downarrow Qe$;                events, advance the time, and return
        $T[j] \ \rightarrow t$;  return $j$;                              its index.

      $\uparrow Qi$;                                             4. Advance to the next bin and repeat.

    $0 \ \rightarrow Qi$,  $Qt0 + Qw \ \rightarrow Qt0$,  $Qt0 + Qw \ \rightarrow Qt1$;           5. Circle back to the first bin.

  return 0;                                                 6. Signal completion of all events.
```

**Algorithm 4. RENUMBER AN EVENT**

**Upon entry to the algorithm, (1)** $n$ contains the new index number, which has no event scheduled. **(2)** $m$ contains the current index number of the event. **At exit, (1)** $n$ is the new index number. **(2)** The event originally scheduled as $m$ is re-scheduled as $n$. Event $m$ no longer has an event scheduled and the index is free to be reused.

```
EventRenumber($n$, $m$)   integer $n$, $m$;
  if $n < 1$ or $n \geq PN$:  $ExitMsg(10)$;                  1. Check the indexes and make sure
  if $m < 1$ or $m \geq PN$:  $ExitMsg(11)$;                     they are in range.

  if $n \neq m$:
    $T[m] \ \rightarrow T[n]$;                                      2. Transfer the time.
    $EventCancel(m)$;                                       3. Cancel the old number.
    $EventSchedule(n,\ T[n])$;                               4. Reschedule as the new number.
```

**Algorithm 5. SORTING**

**Upon entry to the algorithm, (1)** *list* points to an array of forward indexes. *list*[0] is unused. **(2)** $p$ indexes the first element of the list, which ends with a zero. **(3)** $n$ contains the number of items in the list, if known. If zero, the number of items is not known and *sort* should count. **(4)** $c$ compares two list elements $u$ and $v$. It returns negative, zero, or positive when $u < v$, $u = v$, and $u > v$, respectively. **At exit,** *sort* indexes the first element in the sorted list, which ends with a zero. The original ordering is preserved for entries that are equal.

```
integer  $*P,\ pc,\ pr,\ m,\ (*order)(int,\ int)$;

integer $sort(list,\ p,\ n,\ c)$   integer $list[\,],\ p,\ n,\ (*c)(int,\ int)$; integer $i$;
```

```
 $c \to order,\ list \to P$;
```
1. Record calling parameters.

```
 if $n = 0$: $p \to i$; loop while $i > 0$: $P[i] \to i,\ n\uparrow$;
 if $n = 0$ or $p = 0$: return 0;
 if $n = 1$: return $p$;
```
2. Count the number of elements and return empty and single-element lists immediately.

```
 if $n = 2$:
    if $order(p,\ P[p]) \le 0$: return $p$;
     $P[p] \to i,\ p \to P[i],\ 0 \to P[p]$;
     return $i$;
```
3. If the list contains only two elements, sort it by inspection.

```
 $p \to pc$; return $isort(n)$;
```
4. Otherwise sort the full list.

**Partition into sorted sublists. Upon entry, (1)** $n$ defines the minimum number of elements to be sorted. **(2)** $P$ is the list of forward indexes. **(3)** $pc$ indexes the first element of the list. **(4)** *order* compares two list elements. **At exit, (1)** *isort* indexes the first element in the sorted list, which ends with a zero index. **(2)** $m$ defines the number of elements which were actually sorted, greater than or equal to its value on entry. **(3)** $pc$ indexes the element following the last element sorted. If the entire list has been sorted, $pc$ is null.

```
integer $isort(n)$   integer $n$; integer $wp1,\ wp2,\ m1$;
 if $n \le 1$:
    if $pc = 0$: return 0;
     $pc \to wp1,\ 0 \to m$;
```
1. If a single element is requested, initialize variables and check for error in count.

```
     loop : $pc \to pr,\ P[pc] \to pc,\ m + 1 \to m$;
        if $pc = 0$: return $wp1$;
        if $order(pr,\ pc) > 0$:  exit loop;
     $0 \to P[pr]$; return $wp1$;
```
2. Then scan forward in the list to find the longest list that is already in order and return that list.

```
 $isort(n/2) \to wp1$;
 if $n \le m$: return $wp1$;
```
3. If multiple elements are requested, sort the first part of the list and return if enough was sorted.

```
 $m \to m1,\ isort(n - m) \to wp2,\ m + m1 \to m$;
 return $imerge(wp1,\ wp2)$;
```
4. If it was not, then sort what remains and merge the two sublists.

**Merge sublists. Upon entry, (1)** $P$ is the list of forward indexes. **(2)** $p$ and $q$ index the first element of a sorted primary and secondary list, respectively. **(3)** *order* compares two list elements. **At exit,** *imerge* indexes the first element of the list merged in order. In case of equal entries, those from the primary list appear first.

```
integer $imerge(p,\ q)$   integer $p,\ q$; integer $pb,\ v$;
 if $p = 0$: return $q$; if $q = 0$: return $p$;
```
1. Handle empty lists.

```
 if $order(p,\ q) > 0$: $q \to pb,\ 1 \to v$;
 else $p \to pb,\ 3 \to v$;
 loop while $v > 0$:
```
2. Save the beginning of the list and select the proper routine.

```
    loop while $v = 1$: $q \to pr,\ P[q] \to q$;
       if $q = 0$: $-2 \to v$;
       else if $order(p,\ q) \le 0$: $2 \to v$;
    if $v = 2$: $p \to P[pr]$;
```
3. Scan for a secondary element greater than or equal to the current primary element and mend the secondary list.

```
    loop while $v \ge 2$: $p \to pr,\ P[p] \to p$;
       if $p = 0$: $-1 \to v$;
       else if $order(p,\ q) > 0$: $1 \to v$;
    if $v = 1$: $q \to P[pr]$;
```
4. Scan for a primary element greater than the current secondary element, mend the primary list, and repeat.

```
 if $v < -1$: $p \to P[pr]$; else $q \to P[pr]$;
 return $pb$;
```
5. Attach any remaining elements and return the merged list.