

**CDMTCS
Research
Report
Series**

**Bead–Sort: A Natural
Sorting Algorithm**

**Joshua J. Arulanandham
Cristian S. Calude
Michael J. Dinneen**

Department of Computer Science
University of Auckland
Auckland, New Zealand

CDMTCS-170
January 2002

Centre for Discrete Mathematics and
Theoretical Computer Science

Bead–Sort: A Natural Sorting Algorithm

Joshua J. Arulanandham, Cristian S. Calude, Michael J. Dinneen

Department of Computer Science

University of Auckland

Auckland, New Zealand

Email: hi_josh@hotmail.com, {cristian,mjd}@cs.auckland.ac.nz

Abstract

Nature is not only a source of minerals and precious stones but is also a mine of algorithms. By observing and studying natural phenomena, computer algorithms can be extracted. In this note, a simple natural phenomenon is used to design a sorting algorithm for positive integers, called here Bead–Sort. The algorithm’s run-time complexity ranges from $O(1)$ to $O(S)$ (S is the sum of the input integers) depending on the user’s perspective. Finally, three possible implementations are suggested.

1 Beads and Rods

According to Dijkstra [3], “our systems are much more complicated than can be considered healthy, and are too messy and chaotic to be used in comfort and confidence. . . unmastered complexity is at the root of the misery”. One way to cope with this situation is to “go back to Nature” for resources and inspiration (see, for example, [4, 5, 2]), a trend strongly advocated by Rozenberg (see his ideas in the inaugural column “Natural Computing”, [6]; a recent account was presented in [7]).

In this note, a simple natural phenomenon is used to design a sorting algorithm for positive integers, called here Bead–Sort. In what follows, we represent positive integers by a set of *beads* (like those used in an *Abacus*) as illustrated below in Fig. 1.

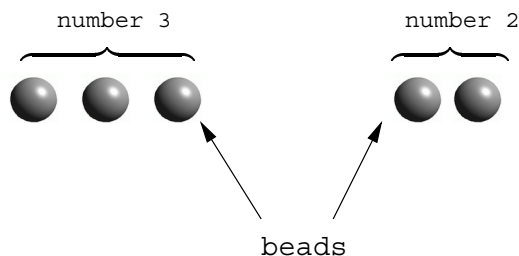


Fig. 1

Beads slide through *rods* as shown in Fig. 2.

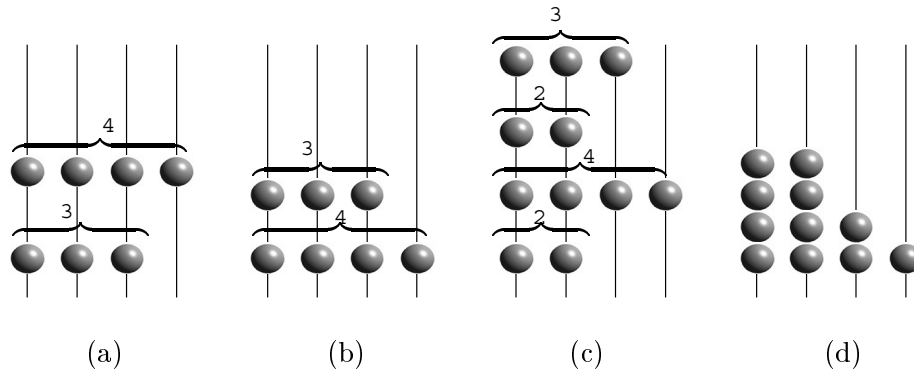


Fig. 2

Fig. 2 (a) shows the numbers 4 and 3 (represented by beads) attached to rods; beads displayed in Fig. 2 (a) appear to be suspended in the air, just before they start sliding down. Fig. 2 (b) shows the state of the *frame* (a *frame* is a structure with the rods and beads) after the beads are ‘allowed’ to slide down. The row of beads representing number 3 has ‘emerged’ on top of the number 4 (the ‘extra’ bead in number 4 has dropped down one ‘level’). Fig. 2 (c) shows numbers of different sizes, suspended one over the other (in a random order). We allow beads (representing numbers 3, 2, 4 and 2) to slide down to obtain the same set of numbers, but in a sorted order again (see Fig. 2 (d)). In this process, the smaller numbers emerge above the larger ones and this creates a natural *comparison* (an online animation of the above process can be seen at [1]).

To study our main procedure, called Bead–Sort, we will adopt a few conventions. *Rods* (vertical lines) are counted always from left to right and *levels* are counted from bottom to top as shown in Fig. 3. A *frame* is a structure consisting of *rods* and *beads*.

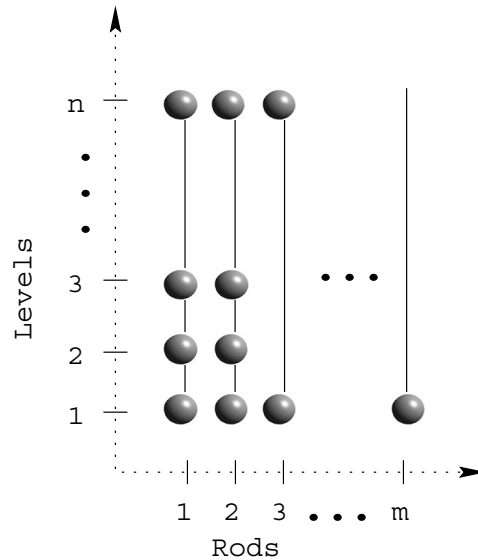


Fig. 3

2 Algorithm and Complexity

In this section, we present the Bead–Sort algorithm and analyze its complexity in three different ways.

Consider a set A of n positive integers to be sorted and assume the biggest number in A is m . Then, the frame should have at least m rods and n levels. The Bead–Sort algorithm is the following:

The Bead–Sort Algorithm

For all $a \in A$ drop a beads (one bead per rod) along the rods, starting from the 1st rod to the a^{th} rod. Finally, the beads, seen level by level, from the n^{th} level to the first level, represent A in ascending order.

The Algorithm *Bead–Sort* is correct. We prove the result by mathematical induction on the number of rows of beads. We claim (*) that the set of positive integers represented by the states of the frame before and after the beads are dropped is the same. Also we claim (**) that the number of beads on each row i , after dropping, is at most the number of beads on row $i - 1$ (the row directly below it).

Consider a set of cardinality $n = 1$. Since there is no possibility for a bead to drop the above two claims hold. Now assume that these two properties hold for an input set whose cardinality is k . Suppose a row $k + 1$ of $m' < m$ beads is now dropped on top of these k rows. Since claim (**) holds there is an index $j \leq m'$ such that all $m' - j$ beads in columns greater than j drop down (from row $k + 1$). If $m' - j = 0$, we are done. Otherwise, the values of row $k + 1$ and row k have been swapped, with any excess beads on row k ready to drop further. Thus, claim (*) holds, after a series of at most k swaps¹. Claim (**) also holds since we repeat these swaps until the force of gravity cannot pull down any more beads.

The time complexity of Bead–Sort can be evaluated at three different coarseness of discretization: complexity (1) treats ‘dropping all beads together’ as a single (simultaneous) operation, complexity (2) treats ‘dropping the row of beads in the frame (representing a number)’ as a distinct operation and complexity (3) treats ‘dropping each and every bead’ as a separate operation.

According to the first measure of complexity, the ‘dropping’ works in one unit of time and hence the complexity is $O(1)$. For the second measure, there are n distinct rows of input beads in the frame and therefore the complexity is $O(n)$. For the third measure, the complexity is equal to the total number of beads dropped. Hence, the complexity is $O(S)$, where S is the sum of the input integers.

Note that in each of the above cases, the number of levels each bead drops is not taken into account. Some implementations might require that this factor is taken into account, since a bead dropping down from a higher level could cost more.

3 An Analog Hardware Implementation

The presence or absence of a *bead* is represented by an analog voltage across electrical resistors. A rod is represented by a series of electrical resistors of increasing value from top to bottom (see Fig. 4).

¹Note that, physically, there are no ‘swaps’, but, the proof assumes the occurrence of ‘pseudo-swaps’, for convenience; the actual algorithm operates in one unit of time to do all these pseudo-swaps.

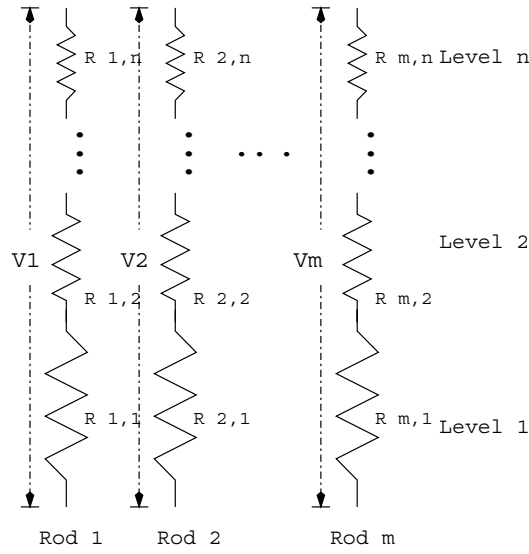


Fig. 4

If the voltage across a resistor is above a preset *threshold-voltage* t , then it indicates the ‘presence of bead’. The values of resistances in series (representing a rod) are arranged in increasing order so that when a voltage is applied across them, there is more ‘concentration’ of charge (voltage-drop) in the bottom resistors than in the top ones; this is the electrical equivalent of the effect of gravity (that attracts beads downward). Let the voltages across the rods be V_1, V_2, \dots, V_m , respectively. Let the currents through the rods be C_1, C_2, \dots, C_m , respectively. Each resistor is denoted by $R_{i,j}$ where i is the rod number and j the level number; $x_{i,j}$ represents the voltage-drop across $R_{i,j}$. The resistors lying on the same level (say, j) have the same value, i.e. $R_{1,j} = R_{2,j} = \dots = R_{m,j}$.

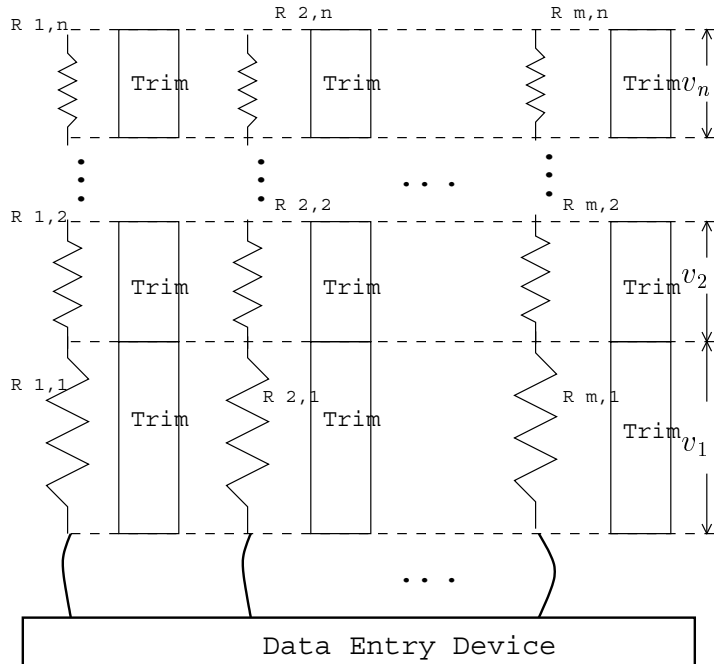


Fig. 5

The voltage across each resistor is fed into a *trimmer* circuit (threshold unit) whose output is given by $x'_{i,j} = 1$, if $x_{i,j} \geq t$, $x'_{i,j} = 0$, otherwise. This is shown in Fig. 5, where dotted lines/dashes do not indicate physical connections. Let v_1, v_2, \dots, v_n denote the sums of individual voltages across resistors (after thresholding) in the 1st, 2nd, \dots , n^{th} levels, respectively (note that in Fig. 5, the detailed circuitry for adding voltages is not shown). Hence we have:

$$\begin{aligned} v_1 &= x'_{1,1} + x'_{2,1} + \dots + x'_{m,1}, \\ v_2 &= x'_{1,2} + x'_{2,2} + \dots + x'_{m,2}, \\ &\vdots \\ v_n &= x'_{1,n} + x'_{2,n} + \dots + x'_{m,n}. \end{aligned}$$

The data-entry device is used to enter data to be sorted. When an integer is keyed in, the equivalent unary representation is generated (for instance, the number 3 is represented as three 1's followed by 0's). The output lines from the data-entry device are attached to the rods consisting of series of resistors as shown in Fig. 5, the first unary digit connected to the 1st rod, and so on. For every '1' in the i^{th} digit of the unary representation, a *voltage increment* δv is applied to the corresponding i^{th} rod (of resistors), i.e. the voltage across the i^{th} rod is increased by δv . This is the way a new *bead* is introduced onto a rod.

Every time a new data is entered, the individual resistors would have a different voltage level; and these voltage levels should reflect the presence/absence of a bead in the frame at that point of time. Hence, the values of the resistors, the *voltage increment* δv and the *threshold-voltage* t should be designed accordingly. After the whole data has been entered, the voltage vector v_1, v_2, \dots, v_n will contain the sorted list (in descending order).

To get a concrete example, consider a simple analog resistor circuit (see Fig. 6) that can sort 3 positive integers, the biggest of which is 3.

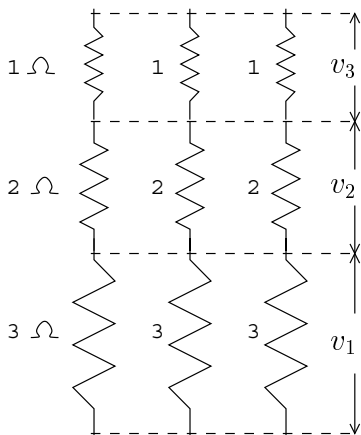


Fig. 6

The circuit has three rods and three levels, 9 resistors on the whole. Let us show how it can sort the data set $\{2, 1, 3\}$. The total resistance in each rod R is $1\Omega + 2\Omega + 3\Omega = 6\Omega$. The *threshold voltage* t is 0.5 volt; the *voltage increment* δv for every new bead is 1

volt. The integers 2, 1, 3 are entered one by one; the corresponding changes in voltage levels across individual resistors and the current after entering the data 2 (110 in unary), 1 (100), 3 (111) are shown in Fig. 7 (a), 7 (b) and 7 (c), respectively. The slanting arrows represent the flow of current and the values of current are shown in amperes. The figures written close to the individual resistors represent the voltage levels across them ($x_{i,j}$); those within the brackets represent the voltage after thresholding ($x'_{i,j}$). The corresponding state of the frame is shown together. Note that after the last integer has been entered, v_1 , v_2 and v_3 represent the sorted list.

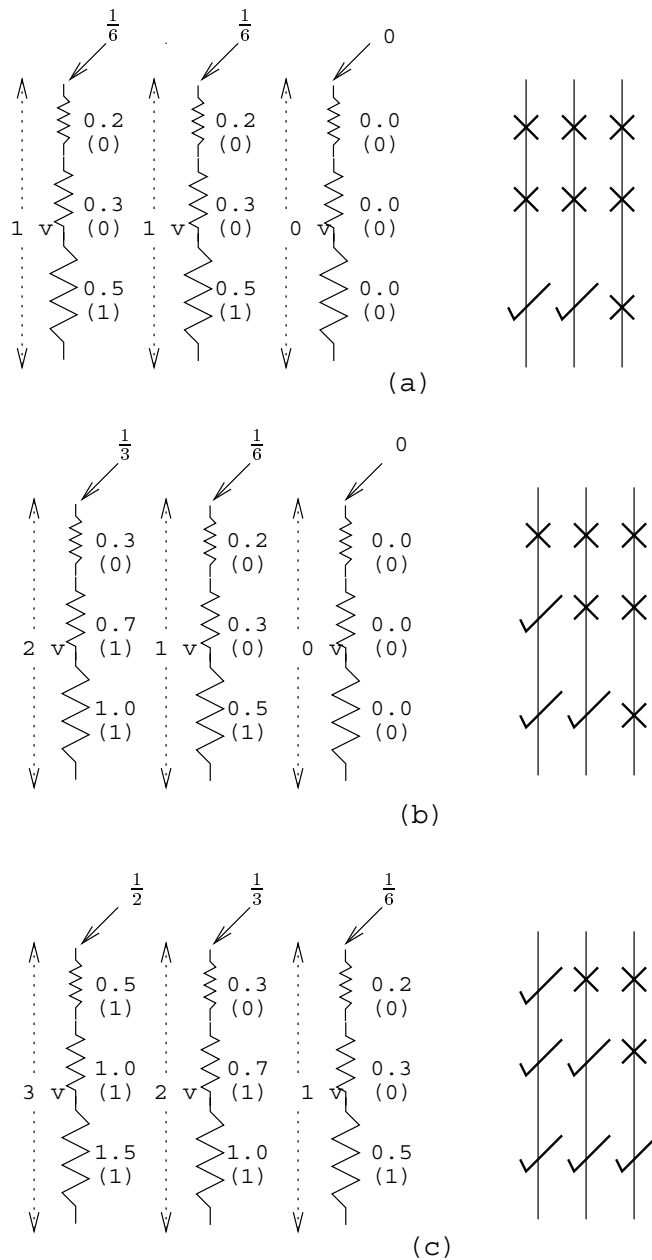


Fig. 7

The time complexity of sorting using the above implementation is due to two components: data entry time and the actual sorting time (time taken for electrical charges

to settle down). In this implementation, data entry and sorting action alternate each other; sorting does not wait till all data have been entered. Let t_1 be the average time taken for entering a single data item and t_2 be the average charge settling-time. It takes $t_1 + t_2$ units for every single data item in the set. The overall complexity is therefore $n \times (t_1 + t_2)$, i.e. $O(n)$.

In the case of manual data entry, t_2 will be so small compared to t_1 such that the actual sorting time (before the next data is entered) is negligible.

4 A Cellular Automaton Implementation

A cellular automaton (CA) is a suitable choice for simulating natural physical systems because it is massively parallel, self-organizing and is driven by a set of simple, local rules (see, for example [8]).

In its simplest form, a CA can be considered a homogeneous array of cells in one, two or more dimensions. Each cell has a finite discrete state. Cells communicate with a number of local neighbors and update synchronously according to deterministic rules. A cell updates its state depending on its current state and its neighbor states.

When modeling physical systems using cellular automata, space is treated as having finitely many locations per unit of volume. Each location is represented by a cell and a state is associated with each cell.

To emulate Bead-Sort, we use a two dimensional CA as shown in Fig. 8.

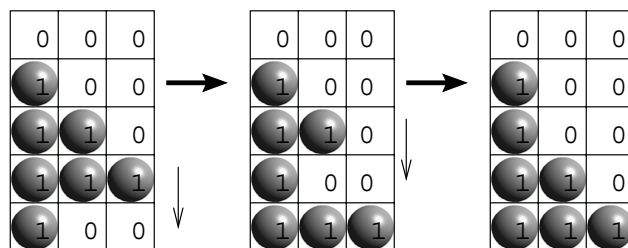


Fig. 8

Each cell has two discrete states. A cell in state ‘1’ represents the space occupied by a bead and a cell in state ‘0’ represents a space without any bead. The local CA rule used to ‘roll down’ a bead into its ‘empty’ lower-neighbor is “a state-1 cell and its state-0 (empty) lower-neighbor cell always swap their states”. Table 1 shows in detail how this rule is applied. The local rules are applied until no change in the configuration can be obtained by any further application. The snap-shots of the CA configurations that evolve while sorting the data-set $\{1, 3, 2, 1\}$ are shown in Fig. 8.

The number of sequential updates of CA configuration necessary for sorting a particular set of data would depend on the ‘degree of disorder’ (entropy) associated with the initial configuration (state) of the frame; this is the extent to which the initial state of the frame ‘deviates’ from the state of a sorted frame (showing the same set of data in sorted form). In the worst case, there will not be more than $n - 1$ sequential updates of CA configuration (note that state-changes in a CA are simultaneous) which leads to a worst-case complexity of $O(n)$.

Present State of Cell x	Upper-neighbour(x)	Lower-neighbour(x)	Next State of Cell x
0	0	0	0
0	0	1	0
0	1	0	1
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	0
1	1	1	1

Table 1

5 A Digital Hardware Implementation

We now discuss a digital implementation: the presence and absence of a bead is represented by the states ‘1’ and ‘0’ of a cell (flip-flop). A rod is represented by an array of cells. Fig. 9 shows how the frame can be described with a two-dimensional array of cells $cell[i, j]$; $cell[i, j] = 1$, if there is a *bead* in the i^{th} rod at the j^{th} level, $cell[i, j] = 0$, otherwise.

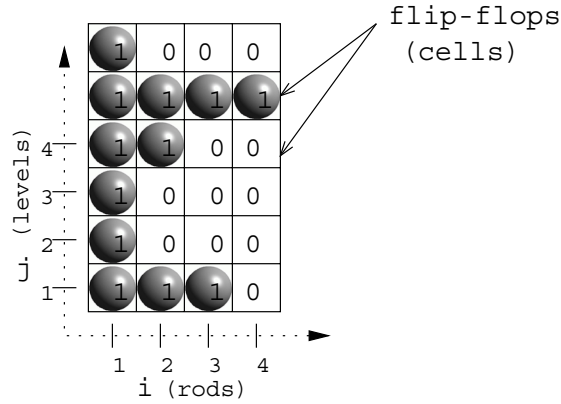


Fig. 9

There are several possible approaches to make beads ‘go down’ one of which is discussed here. The input data (in unary form) is entered one-by-one into a data entry register as shown in Fig. 10. Let the register be represented by a Boolean vector $Bead[i]$; $Bead[i] = 1$ indicates that a bead is to be ‘dropped’ along the i^{th} rod. All 1-bit cells from the register are to be shifted ‘downward’ into the two dimensional array of cells $cell[i, j]$ (representing a *frame*) beneath, before the next data is entered. Thus, after the last data is entered, the whole set would already be sorted in the array of cells.

A simple logic circuit shown in Fig. 10 is connected to each and every cell in the two dimensional array of cells (the frame); when $Bead[i] = 1$, the circuit helps to locate that cell in the i^{th} rod which should ‘receive’ the bead. The circuit attached to that cell alone ‘triggers’ and makes it flip to a 1-state. Thus, shifting the bead downward is quite fast; the 1-bit states in the register (the *beads*) do not have to ‘propagate’ cell-by-cell along the corresponding rods.

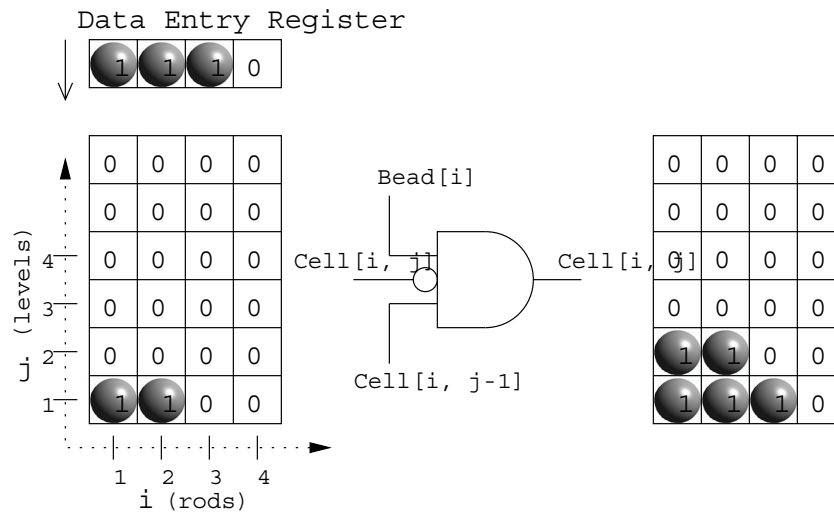


Fig. 10

The circuit attached to each and every cell $cell[i, j]$ enforces the following logic: Set $cell[i, j]$ (make $cell[i, j] = 1$) if (a) it is currently OFF, i.e. $cell[i, j] = 0$ and (b) the cell in the data-entry register representing the i^{th} rod is ON, i.e. $Bead[i] = 1$ (there is a bead up ready to ‘drop down’ along i^{th} rod) and (c) its immediate lower neighbour cell (cell in the next lower level) is ON, i.e. $cell[i, j - 1] = 1$ (which means the bead falling down can not go any further). Note that $cell[i, 0]$ is assumed to be 1. Thus, after the last data is entered, the frame $cell[i, j]$ will have the whole set of data in a sorted form.

The time complexity of sorting using the above implementation is due to two components: data entry time and the actual sorting time (time taken for the cells to switch states). In this implementation also, data entry and sorting alternate each other; sorting does not wait till all data are entered. Let t_1 be the average time taken for entering a single data item and t_2 the average time taken for the cells to switch states. It takes $t_1 + t_2$ units for every single data item in the set. The overall complexity is therefore $n \times (t_1 + t_2)$, i.e. $O(n)$.

The observation made at the end of section 3 is equally valid here.

6 Conclusions

A way to design a natural algorithm is to discover an analogous **natural property** in a physical system, and then to compute by “hitching a ride” on that inherent natural property. In our case, “beads fall down in the sorted order”. Bead-Sort, the *emergent* sorting algorithm, is simple, parallel and low cost. Some possible implementations of Bead-Sort have been discussed. The analog implementation is the most natural implementation; electrical charges behave in the same manner as beads react to gravity. The cellular automaton implementation is simple and very easy to understand. The digital design is easy to implement and is cost effective.

References

- [1] J. J. Arulanandham. *The Bead-Sort*. Animation, www.geocities.com/natural_algorithms/josh/beadsort.ppt.

- [2] C. S. Calude, G. Păun, Monica Tătărâm. A glimpse into natural computing, *J. Multi Valued Logic* 7 (2001), 1–28.
- [3] E. W. Dijkstra. The end of computing science? *Comm. ACM* 44, 3 (2001), 92.
- [4] A. J. Jeyasooriyan. Ball Sort — A natural algorithm for sorting, *Sysreader* (1995), 13–16.
- [5] A. J. Jeyasooriyan, R. Soodamani. Natural algorithms — A new paradigm for algorithm development, In *Proc. 2nd International Conference on Information, Communications and Signal Processing*, CD-Rom, ICICS '99, 1999.
- [6] G. Rozenberg. The natural computing column, *Bulletin of EATCS* 66 (1998), 99.
- [7] G. Rozenberg. The nature of computation and the computation in nature, *International Colloquium on Graph Transformation and DNA Computing*, Technical University of Berlin, 14 February 2002.
- [8] S. Wolfram. *Theory and Applications of Cellular Automata*, World Scientific Publishers, Singapore, 1986.