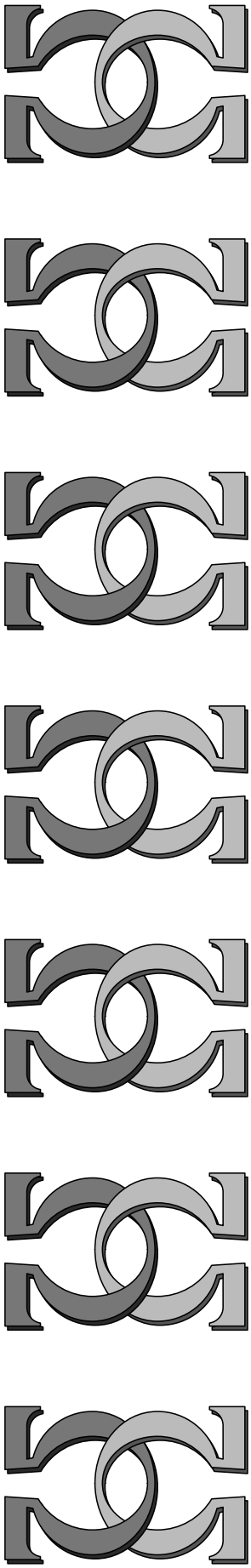

CDMTCS
Research
Report
Series

Rainbow Sort: Sorting at the Speed of Light

Dominik Schultes

Department of Computer Science,
University of Auckland

CDMTCS-244
July 2004

Centre for Discrete Mathematics and
Theoretical Computer Science

Rainbow Sort: Sorting at the Speed of Light*

Dominik Schultes (mail@dominik-schultes.de)
Department of Computer Science, The University of Auckland, New Zealand

Abstract

Rainbow Sort is an unconventional method for sorting, which is based on the physical concepts of *refraction* and *dispersion*. It is inspired by the observation that light that traverses a prism is *sorted* by wavelength. At first sight this "rainbow effect" that appears in nature has nothing to do with a computation in the classical sense, still it can be used to design a sorting method that has the potential of running in $\Theta(n)$ with a space complexity of $\Theta(n)$, where n denotes the number of elements that are sorted.

In Section 1, some upper and lower bounds for sorting are presented in order to provide a basis for comparisons. In Section 2, the physical background is outlined, the setup and the algorithm are presented and a lower bound for Rainbow Sort of $\Omega(n)$ is derived. In Section 3, we describe essential difficulties that arise when Rainbow Sort is implemented. Particularly, restrictions that apply due to the Heisenberg uncertainty principle have to be considered. Furthermore, we sketch a possible implementation that leads to a running time of $O(n+m)$, where m is the maximum key value, i.e., we assume that there are integer keys between 0 and m. Section 4 concludes with a summary of the complexity and some remarks on open questions, particularly on the treatment of duplicates and the preservation of references from the keys to records that contain the actual data. In Appendix A, a simulator is introduced that can be used to visualise Rainbow Sort.

1 Introduction

Traditionally, one makes the distinction between *comparison-based* and *non-comparison-based* sorting algorithms. Comparison-based means that the order of the elements is determined by several comparisons between pairs of elements. It is well known that the lower bound for comparison-based sorting algorithms is $\Omega(n \log n)$, where n denotes the number of elements that are sorted [CLRS01, p. 167]. There are several comparison-based algorithm that are optimal with respect to the asymptotic running time, for example Mergesort and Heapsort [CLRS01, p. 127]. While the running time of these algorithms is $\Theta(n \log n)$, their space complexity is $\Theta(n)$.

Non-comparison-based sorting algorithms do not compare the elements by pairs, but they deal with every single element and move it according to its key. Therefore, these algorithms assume that there are only integer keys in a distinct range from 0 to m. Counting Sort [CLRS01, p. 168] is an example for such an algorithm. Its runtime and its space complexity are $O(n+m)$. The lower bound of $\Omega(n \log n)$ does not apply to non-comparison-based sorting algorithms. In this case, we have only the trivial lower bound of $\Omega(n)$ because, obviously, we have to look at each element at least once if we want to sort it.

In addition to these two traditional types, we can consider *unconventional* sorting methods, which are in many respects so different from the conventional methods that it is appropriate to list them separately. Bead Sort [ACD04] ranks among this third type of sorting methods. In the Bead Sort algorithm each integer x is represented by x beads that are arranged in a horizontal row on vertical rods. When the input is given in this way, the beads fall down until they hit the bottom or another bead that already rests. If each row with x beads is again interpreted as the number x, the rows represent the sorted data after all beads have reached their final position. The

*This paper was written in partial fulfilment of the requirements of COMPSCI 755, Unconventional Models of Computation. It has been accepted for publication in the Journal of Natural Computing.

processing time of Bead Sort is $O(\sqrt{n})$ because the falling beads are accelerated due to gravity so that the duration of the fall is $\sqrt{2h/g}$, where the height h is proportional to n and g is a constant, namely the acceleration of gravity. However, in this case we have to regard the input and output complexity of $\Theta(n)$ as well because it dominates over the processing time. Hence, the total running time is $\Theta(n)$. Unfortunately, the space complexity of Bead Sort is $\Theta(n \cdot m)$, which is a disadvantage. Table 1 summarises the complexities of several sorting methods.

Type	Method	Input	Processing	Output	Σ
comparison-based	Heapsort	n	$n \log n$	n	$\Theta(n \log n)$
non-comp.-based	Counting Sort	n	$n + m$	n	$\Theta(n + m)$
unconventional	Bead Sort	n	$\sqrt{n}$	n	$\Theta(n)$

Table 1: Complexities of several sorting methods

Let us have a look at the gap between the lower and the upper bounds of sorting. For comparison-based sorting, there is no such gap because the upper bounds of known algorithms like Heapsort match with the lower bound of $\Omega(n \log n)$. For non-comparison-based sorting, there is no known conventional algorithm that reaches the trivial lower bound of $\Omega(n)$. However, Bead Sort accomplishes this goal, even though the space complexity of $\Theta(n \cdot m)$ is inconvenient. Hence, there still remains the question whether it is possible to sort in $\Theta(n)$ using a space complexity of less than $\Theta(n \cdot m)$.

2 Idea

2.1 Basic Physical Concepts

Rainbow Sort is based on the basic physical concepts *refraction* and *dispersion*. Here, we give only a brief outline of these concepts as far as it is required in order to be able to describe the idea of Rainbow Sort. For more detailed descriptions the reader may refer to [Hec02].

When a light ray that moves through vacuum enters a transparent material with a refraction index[1] $n > 1$, it gets refracted as illustrated in Figure 1. The extent of the refraction can be expressed by *Snell's Law* [Hec02, p. 101]:

$$\frac{\sin \alpha}{\sin \beta} = \frac{c}{c_n} = n, \tag{1}$$

where the refraction index n corresponds with the ratio of the speed of light c in vacuum to the speed of light c_n in the material with the refraction index n [Hec02, p. 66].

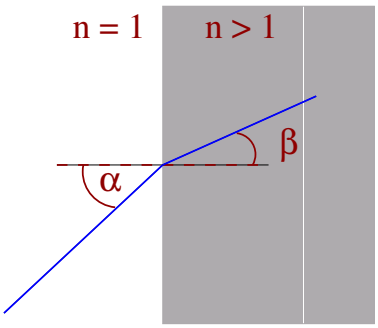

Figure 1: Refraction

When a ray traverses a prism, it is refracted twice, once when it enters the prism and once when it leaves it. The *angular deviation* δ denotes the angle between the incoming and the outgoing ray.

[1] As n is the common identifier of the number of elements *and* of the refraction index, the author decided to use n for both values. The meaning of n will always be clear from the context.

For the symmetrical optical path, i.e., a special case where the ray traverses the prism parallel to the base line, a simple formula for the angular deviation can be derived [Hec02, p. 188]:

$$\delta = 2 \arcsin\left(n \sin\frac{\varepsilon}{2}\right) - \varepsilon,$$

where ε is the angle of the prism opposite to the base line. Figure 2 represents this special case.

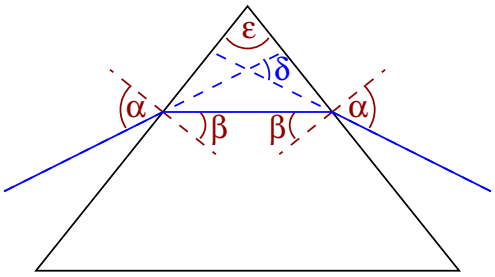

Figure 2: Symmetrical optical path through a prism

Due to the fact that the arcsin-function is strictly monotonic increasing, we can conclude that

$$\text{the greater the refraction index } n\text{, the greater the deviation } \delta. \tag{2}$$

For the general case, the formula gets more complicated, but this fact still holds.

The fact that the refraction index n depends on the wavelength λ of the ray is called *dispersion*. Figure 3 shows the refraction index of crown glass depending on the wavelength.

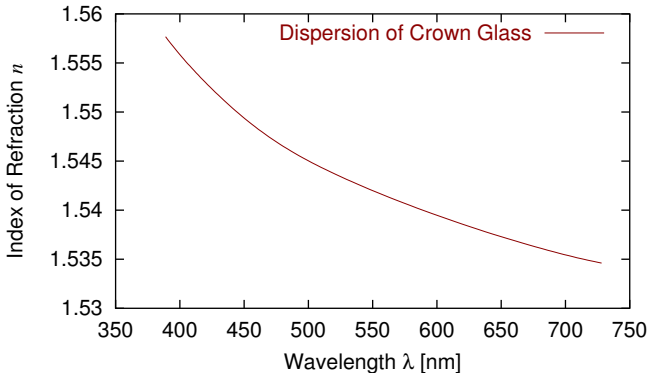

Figure 3: Dispersion of crown glass[2]

We can notice that

$$\text{the less the wavelength } \lambda\text{, the greater the refraction index } n. \tag{3}$$

2.2 Setup and Algorithm

Figure 4 represents the basic setup of Rainbow Sort. It can be divided into an input, a processing, and an output part.

- The input part consists of a light source. The unsorted input data is encoded into wavelengths: each number of the input is mapped to a wavelength. The light source generates a ray whose spectrum consists exactly of the wavelengths that represent the input data.

[2]based on measurement data taken from [Hec02, p. 70]

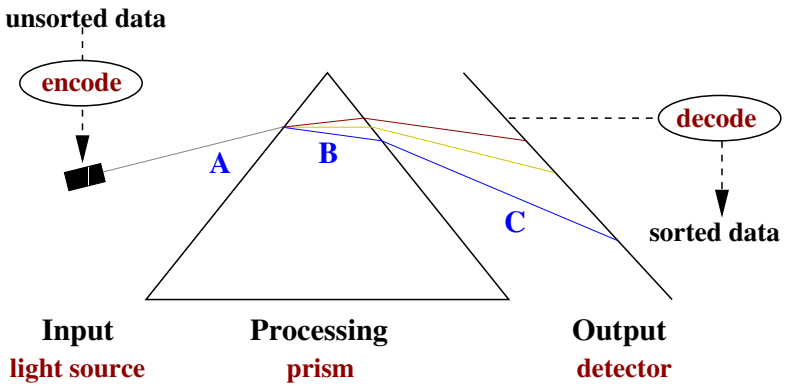

Figure 4: Setup of Rainbow Sort

- The processing part consists of a prism which the generated ray is sent through. Due to the refraction and dispersion, the ray is split into n monochromatic rays that are sorted by wavelength.
- On the output side a detector receives the incoming rays. It is positioned in such a way that the length of the path from the prism is maximal for the minimum wavelength and minimal for the maximum wavelength. The detector decodes the incoming rays and outputs the sorted data.

The basic idea is to use the runtime of the rays to the detector, which increases for decreasing wavelengths, as sorting criterion. Rays with longer wavelengths arrive earlier than rays with smaller wavelengths.

In order to encode the input data, a strictly monotonic increasing function $f : \{0, 1, 2, \ldots, m\} \to [\lambda_{min}, \lambda_{max}]$ is used. It maps from the set of keys to a range of wavelengths that is fixed for any specific implementation. Since the spectrum of light is continuous, we can map to any real number in $[\lambda_{min}, \lambda_{max}]$. Actually, a mapping to rational numbers is sufficient in order to construct such a function f for any given m. For decoding we use the inverse function f^{-1}, which is strictly monotonic increasing, too. So far, this approach cannot deal with duplicates, i.e., with the case that the same number x appears more than once in the input data. Section 4 discusses some possibilities of the treatment of duplicates. However, for now we assume that there are no duplicates for the sake of simplicity.

Figure 5 contains the Rainbow Sort algorithm.

Input
`Ray` $ray := \emptyset$
for each $x \in Input$ **do** $ray := ray \cup f(x)$

Processing
send ray through prism

Output
`Stack` $sorted := \emptyset$
`Wavelength` $cur\lambda := \infty$
whenever $\min \lambda(incoming\ rays) < cur\lambda$ **do**
    $cur\lambda := \min \lambda(incoming\ rays)$
    $sorted$.push$(f^{-1}(cur\lambda))$
    if $sorted$.size $= n$ **then return** $sorted$

Figure 5: Rainbow Sort algorithm

The ray is interpreted as a set of wavelengths (which is initially empty). For each number x of the unsorted input data, x is encoded and the resulting wavelength is added to the ray. In the processing step the ray is sent through the prism. The detection of the sorted wavelengths takes place in a special type of loop: whenever the minimum wavelength of all incoming rays is less than the current wavelength (which is initially set to infinity), the current wavelength is updated and decoded; the resulting number is added to a stack (which is initially empty) that saves the sorted data. Finally, there is a check if all n rays have already arrived. If so, the algorithm terminates and the stack contains the result.

2.3 Correctness and Complexity

In order to prove the correctness of Rainbow Sort, we first show that the wavelengths arrive at the detector in decreasing order: let $\lambda_{min} \leq \lambda_1 < \lambda_2 \leq \lambda_{max}$. Then λ_2 arrives before λ_1.

Proof:

$$\begin{aligned} & \lambda_1 < \lambda_2 \\ \overset{(3)}{\rightsquigarrow}\ & n(\lambda_1) > n(\lambda_2) \\ \overset{(2),(1)}{\rightsquigarrow}\ & \delta(\lambda_1) > \delta(\lambda_2) \quad (\rightsquigarrow \text{ longer path for } \lambda_1) \\ \wedge\ & c(\lambda_1) < c(\lambda_2) \quad (\rightsquigarrow \lambda_1 \text{ slower in the prism}), \end{aligned}$$

where $n(\lambda)$, $\delta(\lambda)$, and $c(\lambda)$ denote the refraction index, the angular deviation, resp. the speed of light in the prism depending on the wavelength λ. Table 2 compares the length of the path and the speed of λ_1 and λ_2.

path / speed	λ_1	λ_2
A	equal / equal	equal / equal
B	longer / slower	shorter / faster
C	longer / equal	shorter / equal

Table 2: Comparison of the length of the path and the speed of two wavelengths $\lambda_1 < \lambda_2$ in the sections A, B, and C as labelled in Figure 4

Before the ray enters the prism (Section A, cp. Figure 4), there is no difference between λ_1 and λ_2. In the prism (B), λ_1 is slower and has to cover a longer distance than λ_2. After the rays have left the prism (C), they move at the same speed, but again λ_1 has to cover a longer distance until it hits the detector. Hence, the whole setup is stacked against λ_1 so that λ_2 arrives first. □

Now, we can complete the *proof of correctness:* The fact that the wavelengths arrive in decreasing order leads to the following observation: whenever a new wavelength arrives, it is smaller than $cur\lambda$. Thus, for each new incoming ray, there is an iteration of the whenever-loop; inside the loop, the new incoming wavelength is assigned to $cur\lambda$. Hence, $cur\lambda$ equals to *all* wavelengths one after the other *in decreasing order*. Therefore, the output is *complete*, i.e., it consists of the same elements as the input, because the encoding function f is injective. Furthermore, we can conclude that the output is *sorted* in decreasing order because the decoding function f^{-1} is strictly monotonic increasing and hence, the existing order is not perturbed by the decoding. □

Table 3 summarises the complexity of Rainbow Sort, independent of a specific implementation.

Input	$\Omega(n)$	$O(?)$
Processing	$\Theta(1)$	
Output	$\Omega(n)$	$O(?)$
Space	$\Omega(n)$	$O(?)$

Table 3: Complexity of Rainbow Sort (independent of a specific implementation). $O(?)$ indicates that a general upper bound cannot be given because it depends on the actual implementation.

The input step has the lower bound of $\Omega(n)$ because the for-loop is executed n times. The runtime of the processing step is $\Theta(1)$, i.e., the actual sorting takes place in constant time because the time interval until all rays have left the prism depends only on λ_{min}, which is a constant. This nice result arises due to the fact that the prism can "process" an arbitrary amount of rays in parallel. The lower bound for the output step is $\Omega(n)$ because the whenever-loop is iterated n times. The space complexity is $\Omega(n)$ due to the output stack that stores all n elements. The upper bounds for the input and output steps and for the space complexity cannot be specified in general since they depend on the actual implementation. Table 4 contains some upper bounds according to the remarks on the implementation that are made in Section 3.

3 Implementation

At this stage, we cannot present a completely elaborated, working implementation of Rainbow Sort. Instead, we point out some essential difficulties that arise when Rainbow Sort is implemented; in particular, the effects of the *Heisenberg uncertainty principle* are discussed in Section 3.1. Furthermore, we make some proposals for a possible implementation. An exceptional property of Rainbow Sort is the fact that the processing step is by far the simplest one; the difficulties appear on the input and output side. Therefore, we concentrate on the input and output parts in the Sections 3.2 and 3.3.

3.1 The Heisenberg Uncertainty Principle

According to the Heisenberg uncertainty principle [SMM97, p. 168], the precision ΔE with which we can know the energy of some system is limited by the time Δt available for measuring the energy. The relation is

$$\Delta E \cdot \Delta t \geq \frac{\hbar}{2} = \frac{h}{4\pi}, \tag{4}$$

where h is Planck's constant. Furthermore, there is the following relation between the energy E of a photon and its wavelength λ [Hec02, p. 57]:

$$E = \frac{hc}{\lambda} \tag{5}$$

Let $E_1 = \frac{hc}{\lambda_1}, E_2 = \frac{hc}{\lambda_2}, \Delta E = E_2 - E_1$, and $\Delta\lambda = \lambda_1 - \lambda_2$. Then $\Delta E = \frac{hc}{\lambda_2} - \frac{hc}{\lambda_1} = \frac{\lambda_1 hc - \lambda_2 hc}{\lambda_1 \lambda_2} = \frac{\Delta\lambda hc}{\lambda_1 \lambda_2}$. Hence, $\Delta E \propto \Delta\lambda$ and, in connection with (4),

$$\Delta t \propto 1/\Delta\lambda. \tag{6}$$

Thus, if we want to determine the wavelength of a ray, we can do so by measuring the energy of the incoming photons and using (5). However, due to (6), we cannot determine the wavelength by a measurement with an arbitrary precision in constant time.

An optimal encoding function f has the following property:

$$\exists\Delta\lambda : \forall x \in \{0, 1, 2, \ldots, m-1\} : f(x+1) - f(x) = \Delta\lambda,$$

i.e., any two adjacent wavelengths have the same difference $\Delta\lambda$, in other words, the range $[\lambda_{min}, \lambda_{max}]$ is evenly subdivided into m intervals of size $\Delta\lambda$ each. Since $\lambda_{max} - \lambda_{min}$ is fixed, we have $m \propto 1/\Delta\lambda$. With reference to (6), this leads to

$$\Delta t \propto m. \tag{7}$$

Therefore, we can conclude that

$$\text{the runtime of an operation that relies on a measurement is } \Omega(m). \tag{8}$$

3.2 Input

In order to realise the light source that is needed on the input side, we can use a laser that can be *tuned continuously* over a range of wavelengths $[\lambda_{min}, \lambda_{max}]$ [Hec02, p. 600]. The tuning can be performed electronically or thermally, i.e., by heating so that the temperature dependence of the system is exploited [KBE+00]. It is difficult to evaluate the time complexity of the tuning process in order to determine the time interval that is required to select the appropriate wavelength with a sufficient accuracy. If we need to measure the energy of the generated ray in order to tune it correctly, the runtime of the input step increases from $\Omega(n)$ to $\Theta(n+m)$ because of (8). This does not exclude that there is a possible realisation that does not rely on a measurement so that a runtime of $\Theta(n)$ is feasible.

In principle, there are two imaginable setups in order to generate a ray whose spectrum consists of n wavelengths. Firstly, we could use n tunable lasers and interlink the generated rays into one single ray. Secondly, we can think of one single tunable laser and some kind of *light storage* that supports operations as used in the algorithm in Figure 5, i.e., it has to be possible to add wavelengths to an existing ray in the light storage and, finally, to emit the ray. While the first proposal is probably realisable with present-day technology, the second one seems rather "up in the air".

3.3 Output

At the detector, the incoming rays are decoded and pushed onto a stack according to the order of their arrival. In order to be able to decode a wavelength, it is required to determine the wavelength first. This is probably the most difficult part of an implementation of Rainbow Sort. In principle, it can be done either

- directly, by measuring the wavelength (or rather the energy), or
- indirectly, e.g., using the property of the chosen setup that each spot at the detector can be assigned to a specific wavelength so that it is possible to conclude the wavelength from the point of contact.

In this section, some concrete ideas regarding the implementation of the detector are discussed.

- The author's first idea was to use a very short pulse, ideally consisting of only one photon for each wavelength, and to measure the energy of the incoming photons in order to determine their wavelengths. The advantage would be that at any point in time only one photon would arrive at the detector so that the detector could not get confused by several rays that are incident simultaneously. Although it is feasible to create very short pulses (in the order of 100 attoseconds = 10^{-16} s) [Ser01], this idea has to be dismissed since such a short pulse cannot be used to measure the energy with a sufficient accuracy because of (4).

- Another approach could be the usage of longer pulses and of a detector that determines the smallest incoming wavelength by a measurement. However, this leads to a runtime complexity of $\Theta(n+m^2)$.

  Proof: Let us assume that the detector is not necessarily a straight line, but it is curved in such a way that the time interval between the arrival of any two adjacent wavelengths is the same Δt.

  Then, we have $T = m \cdot \Delta t$, where T is the time interval between the arrival of λ_{max} and λ_{min}. In connection with (7), we obtain $T \propto m \cdot m$. With reference to the lower bound of $\Omega(n)$ (cp. Table 3), this results in the runtime complexity of $\Theta(n+m^2)$. □

- The measurement of the energy can be avoided if the detector is subdivided into m cells where each cell y corresponds one-to-one with a fixed wavelength $g(y)$ from the range of the encoding function f. When a ray arrives at a cell y, the wavelength of this ray is known to be $g(y)$, without the need of doing a measurement. Hence, the runtime improves to $\Theta(n+m)$.

Proof: The same argument that was used in the previous proof leads again to $T = m \cdot \Delta t$. But in this case, (7) does not apply as no measurement of the energy has to be performed. Hence, Δt need not be adapted when m is increased, but it is constant. Thus, $T \propto m$, which leads to a running time of $\Theta(n + m)$. □

The space complexity of this implementation is $\Theta(n + m)$ as well because in addition to the output stack of size n, we need the m cells of the detector.

There are different ways of collecting information about present wavelengths from the detector. Probably the following method is the most practical one: All cells of the detector are scanned from one end to the other. For each cell y where a ray is (resp. has been) incident, the corresponding number $f^{-1}(g(y))$ is pushed onto the stack that contains the sorted numbers. Obviously, this procedure is linear in the number of cells m so that the derived running time of $\Theta(n + m)$ is not impaired. The advantage of this approach is that it is not required to distinguish between the arrival of the different wavelengths in temporal order, which could be quite difficult in practice since the temporal gaps are very small. Instead, we can wait until the last ray has arrived, and start the scan afterwards. This leads to another advantage: we can simplify the input by using only one laser that sends the encoded numbers one after the other.

In a sense, it is wasteful to scan all m cells, although only n are relevant, where n can be much smaller than m. An improvement of this final step can be useful if a faster implementation of the preceding steps is found. The cells can be connected in an appropriate way with a processor P that manages the output data. When a cell y receives a signal in form of an incoming ray, it sends a message "y" to the processor P, which pushes $f^{-1}(g(y))$ onto the stack. This approach corresponds exactly with the original description of the algorithm in Figure 5. In this case, the crucial part is to ensure that the messages arrive in the correct order and that the system is able to cope with messages that arrive almost simultaneously. This attempt has the potential to improve the running time of the last step to $\Theta(n)$.

Of course, this list is *not* complete. We cannot exclude better implementations that improve the upper bound of the running time and approach the lower bound. For example, it could be worth considering the following idea:

- The measurement of the point in time when a ray arrives at the detector could be used in order to determine the represented number. Let us assume again that the time interval between the arrival of any two adjacent wavelengths is the same Δt. Furthermore, let t_{min} be the point in time when λ_{max} arrives. Then, we can conclude that a ray that arrives at time t represents the number $x = m - (t - t_{min})/\Delta t$. Hence, we "just" have to determine the points of time when the rays arrive and perform this simple calculation each time. In this case, (8) does *not* apply because we are not interested in the energy of the incoming photons, but we only measure the point in time when they arrive.

4 Conclusion

Table 4 combines the results from Section 2 and 3 concerning the complexity of Rainbow Sort. The upper bounds match with those of Counting Sort as mentioned in Section 1. The lower bounds of Rainbow Sort are equal to the trivial lower bounds for sorting. If we use an implementation that relies on measurements of the energy, the lower bound rises and we cannot do better than $\Theta(n+m)$ due to (8). Otherwise, if we can do without a measurement, the complexity of Rainbow Sort will be somewhere between $\Omega(n)$ and $O(n + m)$ depending on the best implementation that can be found.

The optimal treatment of duplicates is an open question. A simple workaround is the adaption of the encoding function f so that each occurrence of the same number x is mapped to a different wavelength. This can be done by extending the domain of f from $\{0, 1, 2, \ldots, m\}$ to $[0, m + 1]$, while the range stays unmodified. When a number x, whose index in the input array is i, is encoded, we take the value of $f_2(x, i) := f(x + i/n)$ instead of just $f(x)$. Therefore, since two

Input	$\Omega(n)$	$O(n+m)$
Processing		$\Theta(1)$
Output	$\Omega(n)$	$O(n+m)$
Space	$\Omega(n)$	$O(n+m)$

Table 4: Complexity of Rainbow Sort. The upper bounds are based on the remarks on the implementation that are made in Section 3.

duplicates have different indices, they are mapped to different wavelengths. However, depending on the actual implementation, this can have a negative impact on the running time because the distance between adjacent wavelengths can get smaller so that we might need more time in order to distinguish between the different wavelengths. For example, let us assume that the first number (index 0) is 1 and the last number (index $n-1$) is 0. As f is strictly monotonic increasing, the difference between $f_2(1,0) = f(1)$ and $f_2(0,n-1) = f(\frac{n-1}{n})$ is smaller than the difference between $f(1)$ and $f(0)$.

In one of the implementations described in Section 3.3, the detector is subdivided into m cells and the original algorithm is modified so that it works in two phases: firstly, the rays are sent one after the other and each cell that is hit stores the fact that a ray has been incident; secondly, all cells are scanned from one end of the detector to the other and the data from the cells are collected. In this case, the treatment of duplicates is simple: instead of storing a boolean value at each cell, we assign a counter to each cell so that it is possible to count the number of incident rays that have the same wavelength.

Another open question is the preservation of the links between the keys and the records that contain the actual data. This is an important issue because practically in every application not only numbers have to be sorted, but keys that are linked to data records. For instance, in a database, whole records consisting of first name, surname and address are sorted by surname.

In order to preserve these links, we have to find a possibility to encode for each key x the address of the corresponding data record $a_x \in A$ into the ray that has the wavelength $f_2(x,i)$, where A denotes the address space.

- Again, this could be done by an appropriate adaption of the encoding function, which would take three parameters (the number x, the index i, and the address a_x) and would map such a triple one-to-one to a wavelength between λ_{min} and λ_{max}. However, again, this could impair the runtime.
- Another imaginable option is to *polarise* the light in order to store additional information.
- Alternatively, the duration of the laser pulses could be varied to encode the addresses of the data records. Let us assume that we have an implementation where the input part is realised by n lasers and the output part is constructed in such a way that the duration of the laser pulses has to be at least t_1 in order to allow precise results. Then, we can select a fixed $t_2 > t_1$ and use an encoding function $h : A \to [t_1, t_2]$ that maps each address a_x one-to-one to a duration $h(a_x)$. Thus, the laser that represents the number x generates a light pulse that has the wavelength $f_2(x,i)$ and the duration $h(a_x)$. The detector can measure the duration t of the light pulse and apply the decoding function h^{-1} in order to determine the address $h^{-1}(t)$ of the record that belongs to the key that is represented by the incoming light pulse.

The latter two approaches could also turn out to be useful with respect to the special treatment of duplicates. For example, the number of occurrences of each number x could be encoded using a function that is similar to h.

A Simulator

A simulator for Rainbow Sort has been implemented in Java. The program, including the source code and a short note on the usage, is available online[3]. The main purpose of the current version of

[3] `http://www.dominik-schultes.de/rainbowSort/`

the simulator is the demonstration of the basic idea, the setup, and the procedure of Rainbow Sort. It is not intended to simulate the physical concepts in every detail, but with some simplifications. The dispersion is implemented by a *linear* interpolation between a minimum and a maximum refraction index, which does not match with the physical reality (cp. Figure 3). Furthermore, the difference between the minimum and the maximum refraction index has been selected to be quite large in comparison with the actual difference. Otherwise, it would be more difficult to follow the optical paths of the different rays.

The simulation is divided into discrete steps. During each step all rays are moved forward according to their current speed. If a ray intersects an edge of the prism, it is refracted according to Snell's Law (1) and its speed is adapted to the material. If a ray intersects the detector, it is decoded and added to the output. Since the time interval of each step is constant, while m is variable, problems can arise for large m when during one step several rays arrive "simultaneously" – at least they seem to arrive simultaneously. As a workaround, all rays that arrive at the same time step are cached, and after each step, the contents of the cache is sorted conventionally and added to the output.

In the simulation only visible light in the range of $[380\text{nm}, 780\text{nm}]$ is used for the obvious reason that invisible light would be a little bit boring for the human user. However, Rainbow Sort is not restricted to visible light, even though the name may suggest this. The optimal choice of λ_{min} and λ_{max} depends on the physical implementation, e.g., on the range of wavelengths that can be generated by tunable lasers, and not on the ability of the human eye.

Acknowledgements

I would like to thank my friend Sebastian Will for his numerous useful comments from the physicist's point of view, which were encouraging and helped me to distinguish between the things that are already feasible with present-day technology, the things that could be realised in the future, and the things that are infeasible because of basic physical principles.

Furthermore, I would like to thank Cristian S. Calude for his review and Joshua J. Arulanandham for his lectures on Bead Sort, which inspired me to think about unconventional methods for sorting. I also wish to thank the referee for the constructive comments and suggestions.

References

[ACD04] J. J. Arulanandham, C. S. Calude, and M. J. Dinneen. A fast natural algorithm for searching. *Theoretical Computer Science*, 320(1):3–13, 2004.

[CLRS01] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms*. MIT Press, 2nd edition, 2001.

[Hec02] E. Hecht. *Optics*. Addison-Wesley, 4th edition, 2002.

[KBE+00] A. Klehr, F. Bugge, G. Erbert, L. Hofmann, A. Knauer, J. Sebastian, V. B. Smirnitzki, H. Wenzel, and G. Tränkle. 300 GHz continuously tunable high power three section DBR laser diode at 1060 nm. In *Proceedings of the 26th International Symposium on Compound Semiconductors*, volume 166 of *Inst. Phys. Conf. Ser.*, pages 383–386, 2000.

[Ser01] R. F. Service. Ultrafast lasers: Strobe light breaks the attosecond barrier. *Science*, 292(5522):1627–1628, 2001.

[SMM97] R. A. Serway, C. J. Moses, and C. A. Moyer. *Modern Physics*. Saunders College Publishing, 2nd edition, 1997.