Computational Models
-draft-

M. Pellegrini*

February 28, 2002

1 Peter W. Shor

Peter W. Shor [Sho97] comments on the historical development of computational models in order to put quantum computational models in perspective.

In the 30's Church, Post and Turing developed computational models to answer the question of which functions are computable and which are not computable. Experience showed that apparently different models yielded the same set of computable functions. Thus the celebrated *Church thesis* which is reported in this paper in the following form:

Thesis 1 (Church) *All computing devices can be simulated by a Turing Machine (TM).*

This is not a mathematical theorem since, while the TM is defined precisely in the language of set theory, the notion of a computing device is not formal but intuitive. However, all attempts at giving a formal definition of the intuitive notion of a computing device up to now did fall under the scope of Church thesis. Church thesis cannot be proved, but it could be disproved if one could produce a counterexample. Moreover there is still the possibility of physical computing devices that are not formalized mathematically and that are not simulated by a TM.

In the 60's and 70's the computer scientist became interested in the following questions, among the computable functions which are feasibly computable and which are non feasibly computable? Here by feasibly computable we mean that the number of steps of the computations grows as a polynomial function of the length of the input. The number of steps of the computation and the length of the input depend heavily on the adopted machine model, but for the classification to be useful we wish feasibility to be independent of the machine model. Experience in simulations of a

*Istituto di Matematica Computazionale del CNR, pellegrini@imc.pi.cnr.it

computational machine by another computational machine showed that all simulations have a polynomial overhead per step, thus leading to the *Quantitative Church thesis* (actually stated by Vergis, Steiglitz and Dickinson in 1986) which is reported in this paper in the following form:

Thesis 2 (Quantitative Church) *Any physical computing device can be simulated by a Turing Machine (TM) in a number of steps polynomial in the resources used by the computing device.*

Sometimes for the purpose of making this thesis stronger the standard TM is augmented with a random number generator. Experience has showed that up to now machines that did violate the Quantitative Church Thesis were not physically feasible. Moreover the unspecified term "resources" can be called upon to solve some case. Typically resources are time and space, but for example in analog computers we may have to consider also "construction precision", or "energy consumption", for quantum computers one has to consider the "precision" of each quantum transformation. In the *Invariance thesis* of van Emde Boas in [vEB90], that reads:

Thesis 3 (Invariance) *Reasonable machines can simulate each other with a polynomial overhead in time and a constant-factor overhead in space,*

it is difficult not to interpret "reasonable" as equivalent to "physically realizable".

It is a fact that computer scientist became convinced of the validity of the Quantitative Church thesis by considering mostly machines realizable in terms of classical physics. This leaves still open the issue whether quantum-mechanics machines may be strictly more powerful than Turing machines with respect to the Quantitative Church thesis. Beinhoff in 1980-1982 proved that quantum-based machines are at least as powerful as TM. At the moment there is no proof of a polynomial overhead simulation of a Quantum Machine (QM) by a TM. For this reason the two models are compared by looking at particular problem. The main result in the paper [Sho97] is that integer factoring can be done on a QM in polynomial time. At the moment it is not known whether integer factoring is solvable in polynomial time on a TM.

2 D. E. Knuth

2.1 Example of a famous algorithm

Knuth [Knu68, pages 1-9] comments on the basic concepts related to the notion of algorithm. After an historical excursus, Knuth recalls that by 1950's the word "algorithm" was associated to Euclid's procedure to find the greatest common divisor of two positive integers, described as follows:
Algorithm E (*Euclid's algorithm*). Given two positive integers m and n, find their greatest common divisor, that is, the largest positive integers that divides both n and m.

E0. [Ensure $m \geq n$] If $m < n$, exchange m and n.

E1. [Find remainder] Divide m by n and let r be the reminder. (Thus $0 \leq r < n$.)

E2. [Is it zero?] If $r = 0$, the algorithm terminates; n is the answer.

E3. [Reduce] Set $m \leftarrow n$, $n \leftarrow r$, and go back to step E1.

∎

This is an example of an algorithm written in Knuth-English style. The algorithm has a characteristic letter E, each step is labeled by this letter followed by an integer so that each step has a different label. Each step has a comment and describes some action that an educated student can perform with the help of a pocket calculator (or paper and pencil). The $\leftarrow$ is the replacement operation and is very important. Labels are used within the "goto" actions. Algorithms are *described* as above, but they are also *performed* (or executed) when a value for m and n is supplied.
A second style of describing algorithms are flowcharts.

2.2 Informal requirements for the notion of algorithm

Algorithm is synonymous with *recipe, process, method, technique, procedure, routine*, except that algorithm has also some subtler feature. Besides being a finite set of rules that give a sequence of operations for solving a specific type of problem, an algorithm has five important features:

1) *Finiteness.* An algorithm must always terminate after a finite number of steps. (When this is not true we speak of computational methods, or reactive processes).

2) *Definiteness.* Each step must be defined precisely. In order to get around ambiguities of natural languages, programming languages have been invented. A computational method expressed in a programing language is called a *program.*

3) *Input.* An algorithm has zero or more inputs. Inputs are taken from specified sets and are supplied before the execution of the algorithm begins, or dynamically as the program is executed.

4) *Output.* A program has one or more outputs, that is, quantities that have a specified relation to the inputs.

5) *Effectiveness.* All its steps must be sufficiently basic so that in principle they can be performed exactly, within a finite time, by an educated person with paper and pencil. Otherwise by a physically realizable device.

Remark of Finiteness. Often finiteness is not enough of a requirement, we want that the number of steps is a finite and small number. Here small is to be understood as bounded by a polynomial in some measure of the input complexity.

2.3 A formal definition of a computational method

The concepts above can be grounded more firmly in set theoretic terms. A computational method is a quadruple (Q, I, Ω, f), where Q represents the set of states, I the set of initial states, Ω the set of final states, and f the computational rules. We have $I \subset Q$, $\Omega \subset Q$, and $f : Q \to Q$. f should leave Ω point-wise fixed (i.e. $\forall q \in \Omega,\ f(q) = q$). Every input $x \in I$ defines a computational sequence as follows:

$$x_0 = x \quad \text{and} \quad x_{k+1} = f(x_k) \text{ for } k \geq 0.$$

The computation terminates in k steps if k is the smallest integer such that $f(x_k) \in \Omega$. If such k exists, x_k is called the output of the computation. An algorithm is a computational method that terminates for every $x \in I$. This definition of algorithm covers points 1) to 4) above, but not point 5) of effectiveness since we have placed little restriction on the choice of f, and no restriction on the nature of the elements of Q.

2.4 Definition of algorithm in terms of a string rewriting system

Example of a definition meeting also requirement 5). Let Σ be a finite set of symbols, and Σ^* the set of all finite strings of symbols from Σ (i.e $\Sigma^* = \bigcup_{k \geq 0} \Sigma^k$). The idea is to use Σ^* to encode the states of Q. More precisely, let N be a positive integer, we define $Q = \Sigma^* \times [0, N]$. The input set is $I = \Sigma^* \times \{0\}$, the output set is $\Omega = \Sigma^* \times \{N\}$. Now we give a definition for f in terms of a string rewriting system (thus the elementary operation involve recognizing sub-strings and overwriting them). Let θ and σ be strings in Σ^*, we say that θ is a *sub-string* of σ if there exist strings α and ω in Σ^* such that $\sigma = \alpha\theta\omega$. For every integer j, with $0 \leq j < N$, we define strings θ_j and ϕ_j in Σ^* and integers a_j, b_j in $[0, N]$, then the following clauses define f:

$$\begin{array}{ll} f(\sigma, j) = (\sigma, a_j) & \text{if } \theta_j \text{ is not a substring of } \sigma. \\ f(\sigma, j) = (\alpha\phi_j\omega, b_j) & \text{if } \alpha \text{ is the shortest string s.t. } \sigma = \alpha\theta_j\omega \\ f(\sigma, N) = (\sigma, N) & \end{array}$$

Note that the component $[0, N]$ makes it clear that one and only one clause of f is applicable at every moment, thus f is indeed a function.

Exercise 1 *Show that the TM (defined below) can be obtained as a specialization of the above definition.*

2.5 The MIX machine language and MIXAL assembly language

In [Knu68, pages 124-164] Knuth gives a very precise description of a register-machine with RA memory and several peripheral units, together with its instruction set (of 64

instructions). The distinction between MIX and MIX assembly language (MIXAL) derives from the standard practice dating from the early days of computing. For each operation a typical cost in time units is specified. This is a variant of the RAM model we will describe shortly.

In [Knu68, pages 186-214] describes programming techniques such as subroutines, co-routines, language emulators, monitor routines.

3 Martin D. Davis and Elaine J. Weyuker

Martin D. Davis and Elaine J. Weyuker [DW85, ch. 2] use the following programming language (called $\mathcal{L}$). We have a denumerable set of Input Variables $X_1, X_2, \ldots$, a denumerable set of Output variables $Y_1, Y_2, \ldots$, and a denumerable set of local variables. Denote with V a generic, X, Y or Z variable. Each variable holds an integer value (or, in different words, each variable is a function $V : \emptyset \to N$, except that we can at different times use the same name to denote a different function). The program is a sequence of instructions of the form:

$$\begin{array}{ll} [\text{label}] & V \leftarrow V + 1 \\ [\text{label}] & V \leftarrow V - 1 \\ [\text{label}] & V \leftarrow V \\ [\text{label}] & \text{if } (V \neq 0) \text{ goto lable} \end{array}$$

Initially all variables are set to zero, except those input variables holding the input values. The state of the program is the Cartesian product of the contents of the variables and the position of the next instruction to be executed. This is a particularly simple computing device; nonetheless the set of functions computable by $\mathcal{L}$ programs coincides with the set of computable functions. Such language is very suitable for discriminating computable and non computable functions. Turing machines are introduced in chapter 6.

4 A. V. Aho, J. E. Hopcrof and J. D. Ullman

Aho, Hopcrof and Ullman [UAH74], introduce several computing devices: Random access machines (RAM), random access stored program machines (RASP), Turing machines (TM), Algol-like language (Algol). RAM and RASP are equipped with two cost models (unit cost and logarithmic cost models). TM have a single natural cost model and Algol relies on naturally understood translation into the RAM model with constant overhead per instruction.

The RAM machine consists of a tape of $X_1, X_2, \ldots$ read only input cells with a read head, a tape of $Y_1, Y_2, \ldots$ write only input cells with a write head, an array of $R_1, R_2, \ldots$ random access cells (called accumulators), and a location counter to point at the current instruction. The Program is a list of instructions, each labeled by its

rank in the list. Register R_0 has a special meaning and is called the accumulator. The instruction set has 12 basic instructions and some instruction have modificators to denote the de-referencing level (i.e. literal constant, direct addressing, indirect addressing). The arithmetic includes addition, subtraction, multiplication and division (but not bit-wise Boolean operations). Note that the program is not stored in memory, thus the location counter does not point to a register location. Examples of an Algol-like higher level language to describe an algorithm together with examples of their translation into a RAM program are given.

Cost models. In the *uniform cost* model each instruction has cost 1. In the *logarithmic cost* model we define the following logarithmic-like function $l(i) = 1$ if $i = 0$, $l(i) = \lfloor \log |i| \rfloor$ if $i \neq 0$. For an instruction (cop, i) the associated cost is $t_{cop} + l(i)$ in the literal constant mode, $t_{cop} + l(i) + l(R[i])$ for the direct addressing mode and $t_{cop} + l(i) + l(R[i]) + l(R[R[i]])$ for the indirect addressing mode. The analysis in the logarithmic cost model requires to keep track of maximum values possible in each register. For example computing n^n by repeated multiplication takes $O(n)$ operations, however, since $l(n^n) \approx n \log n$, the logarithmic cost is $O(n^2 \log n)$, provided that n^n can be stored in a single register.

Random Access Stored Program (RASP). The RASP model is very much like the RAM model except that the program is stored in the registers. This implies that the program can modify itself during its execution. RASP can apply this feature to simulate indirect addressing using direct addressing. It is proved that RAM program can simulate RASP programs and vice versa with constant multiplicative overhead (both in the uniform and logarithmic cost criterion). This is the only textbook where RASP seems to receive some consideration in the context of algorithm complexity analysis literature.

Other models. Other models considered are straight-line programs, logic circuits, decision trees.

Turing Machines. A k-tape TM is composed by k tapes composed of cells, each tape being infinite to the right and endowed by a read/write head. Each cell holds a character from a finite set Σ_t. The input is encoded in a string from the alphabet Σ_i, with $\Sigma_i \subset \Sigma_t$. A particular character $b \in \Sigma_t \setminus \Sigma_i$ is called the "blank". We have a set U of states, and specific states $u_0 \in U$ the initial state, and $u_f \in U$, the final state. The move function is defined as:

$$\delta : U \times \Sigma_t^k \to U \times (\Sigma_t \times \{L, R, S\})^k$$

Initially the state is u_0, the input string is at the beginning of tape 1, followed by all blanks. All other tapes are blank. All heads start in position 1. In one iteration, the state and the characters under the read/write heads are read, using the function δ we decide for each tape, the character to be overwritten and a move of the head: (L)eft, (R)ight or (S)tay. The computation stops when we reach the final state u_f. In the final state the output is a string in the alphabet Σ_i at the beginning of tape 1, followed by blanks, all other tapes are blank, heads in position 1. The time complexity

of the TM is the number of moves used to reach the final state starting with input string x. The storage complexity is the maximum distance traveled by a tape head to the right. It is shown in [UAH74] that multi-tape TM, RAM and RASP under the logarithmic cost model have polynomially related time complexities.
Next in [UAH74, pages 33-39] the syntax of an Algol-like language is given.

5 Peter van Emde Boas

Peter van Emde Boas [vEB90] has written one of the most comprehensive surveys on machine models (64 pages long).

Note that while in the most abstract notion of algorithm, there is hardly the concept of "machine", later on it became customary to distinguish between the "machine" and the "program". Since usually the machine is specified once in a given context, the term algorithm came to indicate only the "program" part. Moreover one could talk about comparing machines since one could map a program under one machine into a program under a second machine by way of step-by-step simulations.

In order to reason about complexity measures such as time and space consumed by algorithms one must refer to the machine model the algorithm is based upon, and a relative cost model.

The existence of many models of computation has not lead to a proliferation of computation theories because all proposed formalism have been shown to be equivalent, in the sense that a computation in one formalism can be simulated by a computation in a second formalism. This has lead to the formulation of Church thesis (Thesis 1).

The study of computational complexity revolves around two families of computational models: Turing Machines and Random Access Models. Since these two large families of models are polynomially related, it is possible to use either of them to discriminate feasible from unfeasible problems. The classes P, NP etc, have equivalent formulations no matter what the machine model is, so that the choice of machine model is for convenience only. This fact has lead to the statement of the Invariance thesis (Thesis 3).

In the study of parallelism and parallel machine models a strong connection between parallel time and sequential storage has been found that justifies the following Parallel computation thesis.

Thesis 4 (Parallel computation thesis) *Any problem solvable in polynomially bounded space on a reasonable sequential machine can be solved in polynomially bounded time on a reasonable parallel machines, and vice versa.*

Sometimes an apparently reasonable model if found out not to be so reasonable after all. In 1974 it was found that the unit-time RAM model with multiplication and bit-wise Boolean operations (MBRAM) was as powerful as a parallel machine,

therefore the MBRAM has been excluded as a reasonable sequential model, and included among the parallel models of computation.

5.1 Formalization of machine models

Machine models can be described in the language of set theory. In these definitions it is common to find a finite object, called the *program*, or *finite control* which operate upon a *memory*. Sometimes the finite control is linked to a *processor*. At any moment in time the finite control is in a particular state, and the number of possible states is finite. Memory is the repository of symbols from a finite alphabet or, in some models, non-negative integers. Memory is arranged in cells with a linear structure (although different patterns exist, e.g. tree-structured memories).The memory is infinite although only a finite portion is used at any time. Programs are written in a programming language. In TM all instruction types are fused in a single read-test-write-move-goto instruction. Special sections of memory are reserved for input and output. A Configuration of the machine includes the state of the finite control and the content of the memory, plus the state of all communication channels between the memory and the finite state controller. The machine realizes a transition between configurations in a time step. A computation is a sequence of configuration transitions starting from an initial configuration up to a final configuration. In *deterministic* machines there is only one next configuration, in *non-deterministic* machines several next configuration are possible. We have *bounded non-determinism* when the number of possible next configuration is a finite number determined only by the machine program (i.e independent of the input).

We can define moreover the concept of *divergent computation*, *termination*, *accepting computation*, *rejecting computation*, *language accepted by a machine*, *language recognized by a machine*, *relation computed by a machine*. So far we had a single finite state control. In parallel machine models we have infinite processor (although only a finite number of them is active at any time) and each has its own finite control.

Given an algorithm A, for every input $x \in I$, we have defined a complexity measure (let us use time in number of steps for definiteness) function $T_A : I \to N$. In order to make predictions we group the input according to the value of the length of their encoding, $|x|$ and we define:

$$T_A^W(n) = \max_{x \in I,\ |x|=n} T_A(x)$$

This gives the *worst case time function* for the algorithm. Using the function $T_A^W(n)$ (and a similar one for space $S_A^W(n)$) we can describe complexity classes (machine dependent). Similarly we have the best case time function:

$$T_A^B(n) = \min_{x \in I,\ |x|=n} T_A(x)$$

and the average case case analysis when we associate to any $x \in I$ a positive probability $p(x)$ so that $\sum_{x \in I} p(x) = 1$, then

$$T_A^A(n) = \sum_{x \in I,\ |x|=n} p(x) T_A(x)$$

5.2 Different types of RAM

In *complexity theory* (i.e. the study of complexity classes) the TM is the standard model. While in the *analysis of algorithms* the standard model is the RAM.

Register Based Machines (a second name for RAMs), have become a standard machine model for the analysis of concrete algorithms. This is due to that fact that register based machine can be thought as abstractions on the structure of commercially produced digital computers. The main point of departure between RAM models and commercial digital computers are (1) infinite word length, versus finite word length, (2) infinite number of registers (or "address space"), versus finite number of registers.

Instructions are divided into 4 categories:

(a) instructions that modify the control flow (conditional and unconditional);

(b) instruction for input/output;

(c) instructions to mode data between accumulators and memory;

(d) instructions to perform arithmetic.

Non-determinism can be achieved with jumping to multiple labels. There are three modes of addressing: literal constants, direct addressing and indirect addressing (resulting in 3 types of load and 2 typed of store instructions).

(a) The weakest model t the SRAM (Successor RAM) where the only allowed arithmetic instruction is the increment by 1.

(b) The (standard) RAM model, allows addition and subtractions.

(c) The MRAM allows allows addition, subtractions, multiplications and divisions.

(d) The MBRAM allows allows addition, subtractions, multiplications, divisions and bit-wise Boolean operations.

Non-deterministic versions are labeled by an initial N. The unit cost model is denoted with U, the logarithmic cost model with L. The weakest model, SRAM, is universal even if indirect addressing is removed, and in this case the model is equal to the model used in recursion theory (see Section 3). The use of indirect addressing

allows to use memory in a sparse fashion but rises two issues: (1) how to initialize the registers to zero? (2) who is going to pay for initialization? Some solution with constant-factor in time and space overhead have been found but, it is not clear however how much these solutions are measure-independent.

All models and all cost measure are polynomially related except the MRAM-UTIME that is exponentially related to the MRAM-LTIME (for this reason the MRAM with Uniform cost measure is often excluded from the reasonable sequential models in complexity theory). Suppose that in the uniform cost model we allow multiplication and bit-wise operations on words of at most k-bits; by using table-look up techniques and $O(1)$ combination steps we can quickly simulate unrestricted multiplication operations [SS90].

In [vEB90, pages 27-32] it is discusses how a TM and a RAM are related polynomially both in time and is space complexity. Here two subtle issues are (1) that also the address of a register has to be charged to the space cost of that cell, and (2) how to handle sparse memory allocation.

In [vEB90, pages 32-38] two additional models are introduced, the *pointer model* (also known as Storage modification machines) where the memory is modeled as a rooted, edge-labeled graph, and the uniform and non-uniform circuit models.

In [vEB90, pages 32-61] parallel models of computation are discussed.

6 Discussion on the bounded word RAM model

We have seen in Section 5 that unit time RAM models with multiplication must be handled with care even if we limit the word size of factors. In particular there exist computations under the MRAM model that have an exponential gap between the unit and logarithmic cost models. Naturally this does not imply that every computation has an exponential gap between unit and logarithmic cost model. It becomes interesting thus to study specific important problems (an specifically their lower bound in a particular machine-cost model). It is well known that sorting n integers in the comparison model (i.e. we have a RAM model but no arithmetic, only comparison and conditional jumps.) requires $\Omega(n \log n)$ operations. J. Paul and W. Simon [PS80] have shown that such lower bound holds even in the unit cost RAM model with addition, subtraction, multiplication, comparison with 0 (but neither division nor bit-wise Boolean operations). On the other hand if we have addition, subtraction, multiplication, comparison with 0, division and bit-wise Boolean operations, then it is possible to sort in time $O(n)$. This algorithm however handles intermediate number that are $O(n^2)$ times longer than the longest input number, thus abusing the unit cost assumption. The problem involving sorting and searching of integers become interesting in a new model the *Bounded Word RAM* model where each input and each intermediate result is assumed be a non-negative integer smaller than 2^b, thus fitting in a b-bits word size register. In this setting we cannot have more than 2^b distinct input numbers thus the complexity bounds will be functions $T(b, n)$ of b and $n < 2^b$.

The seminal paper on the subject is [FW90]; the large literature that has come out of this model is cited in [Wil00].

7 Conclusion on oimputational Model for analysis.

Unit cost RAM models with multiplication and other operations are easy to use and permit an analysis of rather complicated algorithms. On the other hand they are also dangerous since it is possible to devise algorithms that pay cost one for many operations (thus actually simulating a parallel computer).

The conclusion that can be drawn is that such model is relatively safe, in the sense that such analysis provide realistic prediction of the behaviour of a commercial computer, provided that neither table lookup nor unusually long numbers appear in the execution of the algorithm.

8 On the Concept of "Data Structure"

Most of the discussion on the machines and models of computation in the previous Sections was aimed at describing the function f in 2.3. On the other hand the concept of state was not so elaborated upon. In a further refinement in sub-section 2.4 the state was composed of a finite part and a potentially infinite one $Q = \Sigma^* \times [0, N]$. The infinite one is called memory. In TM the memory was the content of the (multi)-tape, in RAM the content of the (infinite) array of registers. The design of "Data Structures" is the art and craft of organizing information in memory so to speed up the execution of algorithms. From now on we discuss only RAM memory. The interplay between the index of a register, its content and indirect addressing is essential. If only direct addressing is allowed, any RAM program is independent of permutations on the space of addresses (i.e. renaming of the variable in the program) and therefore we cannot use the index of a register to encode any information. Thus indirect addressing is essential to obtain the feature that the position of the data give us some information on the data itself. For example, if we store n distinct numbers *in increasing order of magnitude* in *consecutive* registers $R[x], R[x+1], .., R[x+n]$ then we have structural information we can exploit; for example we know that the median element is in location $R[x+n/2]$ (for n even) and all elements in $R[x], ..., R[x+n/2-1]$ are smaller than the element in $R[x+n/2]$.

References

[DW85] M. D. Davis and E. J. Weyuker. *Computability, Complexity and Languages.* Academic Press, New York, 1985.

[FW90] M. L. Fredman and D. E. Willard. BLASTING through the information theoretic barrier with FUSION TREES. In *Proceedings of the 22nd Annual ACM Symposium on the Theory of Computing*, pages 1–7, Baltimore, MY, May 1990. ACM Press.

[Knu68] D. E. Knuth. *The Art of Computer Programming I: Fundemental Algorithms*. Addison–Wesley, Reading, Massachusetts, 1968.

[PS80] J. Paul and W. Simon. Decision trees and random access machines. In *Smposium uber Logik und Algorithmik*, 1980. See also K. Mehlhorn, Sorting and Searching, pp. 85–97, Springer-Verlag, Berlin, 1984.

[Sho97] Peter W. Shor. Polynomial-time algorithms for prime factorization and discrete logarithms on a quantum computer. *SIAM Journal on Computing*, 26(5):1484–1509, 1997.

[SS90] J.P. Schmidt and A. Siegel. The spatial complexity of oblivious k-probe hash functions. *SIAM Journal on Computing*, 19(5):775–786, 1990.

[UAH74] J. D. Ullman, Alfred V. Aho, and John E. Hopcroft. *The Design and Analysis of Computer Algorithms*. Addison-Wesley, Reading, 1974.

[vEB90] P. van Emde Boas. Machine models and simulations. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science: Volume A, Algorithms and Complexity*, pages 1–66. MIT Press, Cambridge, MA, 1990.

[Wil00] Dan E. Willard. Examining computational geometry, Van Emde Boas trees, and hashing from the perspective of the fusion tree. *SIAM Journal on Computing*, 29(3):1030–1049, June 2000.