FAST EXPONENTIATION USING THE TRUNCATION OPERATION

NADER H. BSHOUTY, YISHAY MANSOUR, BARUCH SCHIEBER AND PRASOON TIWARI

Abstract. Given an integer $k$, and an arbitrary integer greater than $2^{2^k}$, we prove a tight bound of $O(\sqrt{k})$ on the time required to compute $2^{2^k}$ with operations $\{+, -, *, /, \lfloor \cdot \rfloor, \leq\}$, and constants $\{0, 1\}$. In contrast, when the floor operation is not available this computation requires $\Omega(k)$ time. Using the upper bound, we give an $O(\sqrt{\log n})$ time algorithm for computing $\lfloor \log \log a \rfloor$, for all $n$-bit integers $a$. This upper bound matches the lower bound for computing this function given by Mansour, Schieber, and Tiwari. To the best of our knowledge these are the first non-constant tight bounds for computations involving the floor operation.

Subject classifications. 68Q40.

1. Introduction

Suppose that an integer $k$, and an arbitrary integer greater than $2^{2^k}$ are given as inputs. We present a tight bound of $O(\sqrt{k})$ on the time required to compute $2^{2^k}$ using operations from the set $\{+, -, *, /, \lfloor \cdot \rfloor, \leq\}$ and constants $\{0, 1\}$. In contrast, when the floor operation is not available, and available constants are restricted to the set $\{0, 1\}$, this computation requires $\Omega(k)$ time.

We apply the upper bound to get the following algorithms: (1) an algorithm for computing $\lfloor \log \log a \rfloor$, for all $n$-bit integers $a$, in $O(\sqrt{\log n})$ time; and (2) an algorithm for deciding whether an integer $a$ in some range $[2^{2^k}, 2^{2^k}+1)$ (where $k$ is unknown) is a perfect square (i.e., is a square of another integer), in $O(\sqrt{\log \log a})$ time. Both upper bounds match respective lower bounds given in [8].

Most of the earlier work on the complexity of arithmetic computations has been in models that do not allow the truncation operation. Only a handful of papers considered models that do allow the truncation operation, among

1Throughout this paper "/" stands for exact division, e.g., $8/5 = 1.6$. We remark that given the other operations, $\lfloor \cdot \rfloor$ and mod can simulate each other in $O(1)$ operations.
them are \([4; 3; 5; 2; 7; 8]\). (See [7] for a detailed discussion of earlier work.) However, the truncation operation is widely available on computers. Thus, it is reasonable to ask whether the floor operation strengthens the computational power significantly. An affirmative answer to this question has been known for some time. For example, consider the problem of deciding whether an \(n\)-bit integer is even or odd. Using the floor operation this can be done in \(O(1)\) time. On the other hand, Stockmeyer [10] proves a \(\Theta(n)\) bound on the time required for this computation with operations \{+, -, \*, /, \leq\}. Here, we give additional evidence of the computational power of the floor operation. Perhaps the highlight of the paper is the non-trivial manner in which the floor operation is used to construct optimal algorithms.

The computation model used for the upper bounds is the standard RAM model (see, e.g., [9; 1]). The only exception is the model used for our second application, where we do not bound the size of the numbers involved in the computation. Notice that since we are not using the indirect addressing feature of the RAM model, the results apply also to the computation tree model (see, e.g., [11; 8]). The computation model used for the lower bound is the computation tree model, but the same lower bound extends to RAMs with indirect addressing using the technique given in [8].

The rest of the paper is organized as follows. In Section 2 we describe the fast exponentiation algorithm. In Section 3 we give the applications of our exponentiation algorithm. In Section 4 we prove the matching lower bound. For completeness, we also describe the \(\Omega(k)\) lower bound for computing \(2^{2^k}\) when the floor operation is not included.

### 2. The exponentiation algorithm

Intuitively, it seems that the fastest way to compute \(2^{2^k}\) is by successive squaring. This gives an \(O(k)\) time procedure for computing this number. Surprisingly, as shown below, using the floor operation and an additional input \(a > 2^{2^k}\), the integer \(2^{2^k}\) can be computed in \(O(\sqrt{k})\) time.

First, we give the algorithm for the case \(k = m^2\), for some integer \(m\). Later, we generalize the algorithm to arbitrary \(k\).

**Theorem 2.1.** Given an integer \(m\), and an arbitrary integer \(a\) greater than \(2^m^2\), the integer \(2^{2^m^2}\) can be computed in \(O(m)\) time using only constants \(\{0, 1\}\), and operations from the set \{+, -, \*, /, \lfloor \cdot \rfloor, \leq\}.

**Proof.** Let \(a_{1,0} = 2a\), and compute \(a_{1,i} = a_{1,i-1}^2\), for \(i = 1, 2, \ldots, m\), by successive squaring. This requires \(m\) multiplications. Note that \(a_{1,i} = (2a)^{2^i}\).