On the complexity of the $2^k$-ary and of the sliding window algorithms for fast exponentiation.

Ottavio G. Rizzo

1st March 2004

We present the $2^k$-ary and the sliding window algorithms for fast exponentiation. We give a precise formula for the error terms of their complexity and we discuss how to choose the parameters or the exponent optimally.

Key words: binary powering, addition chain, window method

MSC2000: 11Y16

Let $E$ be an (additive) monoid and suppose we want to compute $nP$ where $n \in \mathbb{N}$ and $P \in E$. The obvious way, summing $n$ copies of $P$, is not very efficient: it requires $n - 1$ additions. For example, $12P = 2(2(2P + P))$ uses only one addition and three doublings, rather than eleven additions.

The problem of computing a scalar multiple (or the exponent, in a multiplicative setting) using few additions/doublings is the same as finding an addition chain of small length. We recall that an addition chain for $n \in \mathbb{N}$ is a sequence $a_1 = 1 < a_2 < \cdots < a_\ell = n$ such that $a_i = a_j + a_k$ with $j, k < i$ for every $i = 2, \ldots, \ell$ (see [9] for a very thorough introduction). Finding the shortest addition chain for a given integer $n$ is commonly considered to be a very hard problem (although it has not been proved it is NP-hard, as often – but erroneously – reported in the literature referring to [5]: see the remarks in [1] about this misconception); so in many applications it is important to find in a reasonable time a chain which is reasonably short.

A few words on what “reasonably short” means: let $\lambda(n) = \lfloor \log_2(n) \rfloor$; using Algorithm 4, we can find an addition chain for $n$ of length $\lambda(n) + (1 + o(1)) \lambda(n)/\lambda(\lambda(n))$. On the other hand, Erdős proved in [6] that, asymptotically, for almost every $n$, the shortest addition chain for $n$ has length $\lambda(n) + \lambda(n)/\lambda(\lambda(n))$.

In practical applications, there is one further constraint: one cannot store too many intermediate steps! In the simplest case, we allow only the two following operations: $a_i = 2a_{i-1}$ or $a_i = a_{i-1} + 1$. Such an algorithm, as the one we are presenting in the first section, needs to store just two values. We can improve the algorithm (i.e., shorten the
length of the chain) by precalculating some values: if $\mathcal{P} \subset \mathbb{N}$ is a finite subset, we say that a given addition chain is based on $\mathcal{P}$ if it involves only operations of the form $a_i = 2a_{i-1}$ or $a_i = a_{i-1} + a_j$ with $a_j \in \mathcal{P}$. Clearly, one has to store $1 + \# \mathcal{P}$ values in order to execute the algorithm.

As a final remark, we will not try to give attributions for the algorithms (a variant of Algorithm 1 was already known to Egyptian mathematicians in 1800 B.C.!) the interested reader may consult [1], which provides an excellent bibliography. Furthermore, the main terms of our theorems are of course known (see, for example [4] for Theorem 7; the main term of Theorem 15 appears in [10], although the error there is $O(1)$): our goal for this paper is to give a description as good as possible of the small terms. At last, we have been informed after our presentation that H. Cohen gives similar results (without a proof) in [3] about the sliding window algorithm.

1 The left binary algorithm

Given $n \in \mathbb{N}$, let $n = \sum_{i=0}^{\lambda} n_i 2^i$ be its binary expansion, where $n_i \in \{0,1\}$, $n_\lambda = 1$.

Algorithm 1. The left binary algorithm.

1. let $n = (n_\lambda n_{\lambda-1} \cdots n_0)_{2}$, $Q = P$
2. while $i = (\lambda - 1) \cdots 0$:
   3. let $Q = 2Q$
   4. if $n_i = 1$: let $Q = Q + P$
5. return $Q$

We want to analyse the complexity of the algorithm, under the assumption that every operation different from the doubling and the addition of points in $E$ is negligible in terms of time and memory. In particular, we are interested in the average number of doublings and additions when $n$ runs over all integers of exactly $e$ bits, that is $\lambda(n) = e - 1$. Let $c_D(n)$, resp. $c_A(n)$, be the number of doublings, resp. additions, that the algorithm requires for a given $n \in \mathbb{N}$, not including precomputations. If $\mathcal{P}$ is the set of values that need to be precomputed, let $C_D(\mathcal{P}) = \#\{\text{doublings required by } \mathcal{P}\}$,

$$C_D^0(e) = \frac{1}{2^e-1} \sum_{\lambda(n) = e-1} c_A(n), \quad C_D(e) = C_D^0(e) + C_D(\mathcal{P}).$$

Analogously for $C_A^0(e)$, $C_A(\mathcal{P})$ and $C_A(e)$. It is clear that, in the case of Algorithm 1, $c_D(n) = \lambda(n)$ and $c_A(n) = w(n) - 1$, where $w(n) = \#\{n_i \neq 0\}$ is the Hamming weight of $n$. Since $\mathcal{P} = \{1\}$, i.e. there are no precomputations, $C_D(\mathcal{P}) = C_A(\mathcal{P}) = 0$. It is then a simple exercise on induction (but cf. the proof of Theorem 7) to prove that:

Theorem 2. For Algorithm 1: $C_D(e) = e - 1$ and $C_A = (e - 1)/2$ for any $e \in \mathbb{N}$.

The binary algorithm can been improved in many ways, but the number of doublings is (essentially) unavoidable. The variants we will present will reduce the number of additions, at the cost of precomputing (and storing!) some more values; in other words, given Erdős’s estimate, the goal is to have $C(\mathcal{P}) + C_A(e) \sim \lambda(n)/\lambda(\lambda(n))$ for $n \gg 1$. 

The $2^k$-ary algorithm

Let $k$ be a positive integer, which we will usually suppose $> 1$. The idea is to write $n$ in the base $2^k$: $n = \sum_{i=0}^{\ell} n_i 2^{ki}$ where $n_i \in \{0, \ldots, 2^k-1\}$, $n_\ell \neq 0$ and $\ell = \lfloor \lambda(n)/k \rfloor$.

**Algorithm 3.** The $2^k$-ary algorithm, simple form.

```plaintext
while $i = 2, \ldots, 2^k - 1$: store $iP$
2 let $n = \sum_{i=0}^{\ell} n_i 2^{ki}$, $Q = n_\ell P$
while $i = (\ell - 1) \ldots 0$:
4 let $Q = 2^k Q$
5 if $n_\ell \neq 0$: let $Q = Q + n_\ell P$
6 return $Q$
```

Notice that $C_D(P) = 1$, $C_A(P) = 2^k - 3$ and $2^k - 2$ multiples of $P$ need to be stored. We can cut these requirements in half if we note that we do not really need to store the even multiples:

**Algorithm 4.** The $2^k$-ary algorithm, better form.

```plaintext
while $i = 3, 5, \ldots, 2^k - 1$: store $iP$
2 let $n = \sum_{i=0}^{\ell} n_i 2^{ki}$, let $n_i = \nu_i 2^{\varepsilon_i}$ // where $\nu_i$ is odd and $\varepsilon_i = 0$ if $n_i = 0$
3 let $Q = \nu_\ell$; let $Q = 2^{\varepsilon_\ell} Q$
while $i = (\ell - 1) \ldots 0$:
5 let $Q = 2^{k-\varepsilon_\ell - 1} Q$
6 if $n_\ell \neq 0$: let $Q = Q + \nu_\ell P$
7 let $Q = 2^{\varepsilon_\ell} Q$
8 return $Q$
```

Let $\varepsilon(n) = \varepsilon_\ell(n)$; i.e., $\varepsilon(n) = \nu_2(\lfloor (n/2^{k\ell}) \rfloor)$, where $\nu_2$ is the 2-adic valuation. It is clear that $c_A(n)$ is the same for both algorithms. On the other hand, Algorithm 4 will require exactly $\varepsilon(n)$ more doublings than Algorithm 3. Although in practise only Algorithm 4 would be used, we will study the complexity of its simpler version and use the following proposition to deduce its own complexity.

**Proposition 5.** Write $i \mod k$ for $i - \lfloor i/k \rfloor k$. Then, for every integer $e \geq 1$:

$$\frac{1}{2e-1} \sum_{\lambda(n)=e-1} \varepsilon(n) = 1 - 2^{-((e-1) \mod k)}.$$

**Proof.** If $n_\ell = \lfloor n/2^{k\ell} \rfloor$ and $n' = n - n_\ell 2^{k\ell}$ then

$$n' = n - \left\lfloor \frac{n}{2^{k\ell}} \right\rfloor 2^{k\ell} = \left( \frac{n}{2^{k\ell}} - \left\lfloor \frac{n}{2^{k\ell}} \right\rfloor \right) 2^{k\ell} \in [0, 2^{k\ell}) ,$$

and

$$\lambda(n_\ell) = \left\lfloor \log_2 \left( (n - n') 2^{-k\ell} \right) \right\rfloor = \left| \log_2 (n - n') \right| - \ell k = e - 1 - \ell k;$$
Write $\tilde{e}$ for $e - 1 - \ell k$. Then, thanks to Lemma 6,

$$\sum_{\lambda(n)=\tilde{e}} \varepsilon(n) = \sum_{\lambda(n) = \tilde{e}} \sum_{n' = 0}^{2^{\ell k} - 1} \varepsilon(n'2^{\ell k} + n') = \sum_{\lambda(n) = \tilde{e}} \sum_{n' = 0}^{2^{\ell k} - 1} \varepsilon(n'2^{\ell k})$$

$$= 2^{\ell k} \sum_{\lambda(n) = \tilde{e}} v_2(n') = 2^{\ell k}(2^{\tilde{e}} - 1) = 2^e - 2^\ell k.$$

The proposition follows immediately, once we notice that $\tilde{e} = (e - 1 \mod k)$. \hfill $\lozenge$

**Lemma 6.** For every integer $e \geq 0$ we have $\sum_{\lambda(n)=e} v_2(n) = 2^e - 1$.

**Proof.** Denote $\theta(e)$ such a sum. Clearly, $\theta(0) = v_2(1) = 0$. Suppose now that $e > 0$; if $n$ is odd, then $v_2(n) = 0$, thus

$$\theta(e) = \sum_{\lambda(n)=e} v_2(n) = \sum_{\lambda(n')=e-1} v_2(2n') = \sum_{\lambda(n')=e-1} 1 + v_2(n') = 2^{e-2} + \theta(e-1).$$

It is straightforward to verify that $\theta(e) = 2^e - 1$ satisfies these inductive properties. \hfill $\lozenge$

Notice that in Algorithm 3 additions occur only in line 5, and only when $n_i \neq 0$ with $0 \leq i < \ell$. Thus, if $w_k(n) = \#\{n_i \neq 0\}$, where $n = \sum n_i2^{k_i}$, the number of additions is $c_A(n) = w_k(n) - 1$, for every $n > 0$. In the same algorithm, $k$ doublings are executed each time line 4 is run, which happens $\ell$ times; thus $c_D(n) = \ell k$. Therefore, considering Proposition 5, averaging, and applying Lemma 8, we get:

**Theorem 7.** If $e > k$, we have for Algorithm 4:

$$C_A^0(e) = \left\lfloor\frac{e - 1}{k}\right\rfloor (1 - 2^{-k}), \quad C_D^0(e) = \left\lfloor\frac{e - 1}{k}\right\rfloor k + 1 - 2^{-((e-1) \mod k)};$$

$$C_A(\mathcal{P}) = 2^{k-1} - 1 \quad C_D(\mathcal{P}) = 1.$$

**Lemma 8.** If $1 \leq e \leq k$, then $\frac{1}{2^{e-1}} \sum_{\lambda(n)=e-1} w_k(n) = 1$; while if $e > k$,

$$\frac{1}{2^{e-1}} \sum_{\lambda(n)=e-1} w_k(n) = 1 + \left\lfloor\frac{e - 1}{k}\right\rfloor (1 - 2^{-k}).$$

**Proof.** Write $\omega(e)$ for such a sum. Since $w_k(n) = 1$ for every $n \in \mathbb{N}$ with $\lambda(n) < k$, we get $\omega(e) = 1$ for every $e$ such that $1 \leq e \leq k$.

If $n > 2^k$, write $n = n_\ell 2^{k+\ell'}$ as in the proof of Proposition 5; then $w_k(n) = 1 + w_k(\ell')$. Thus, if $e > k$ and $\tilde{e} = e - 1 - \ell k$,

$$2^{e-1} \omega(e) = \sum_{\lambda(n')=\tilde{e}} \sum_{n' = 0}^{2^{\ell k} - 1} 1 + w_k(n') = 2^{e-1} + 2^\ell \sum_{n' = 0}^{2^{\ell k} - 1} w_k(n').$$

It is an easy exercise to check that $\sum_{n=0}^{2^k-1} w_k(n) = \ell(2^k - 1)2^{k-1}$. Hence, $\omega(e) = 1 + \frac{1}{2^e} \cdot 2^k \cdot \ell(2^k - 1) + \frac{1}{2^e} \cdot \left\lfloor\frac{(e-1)}{k}\right\rfloor (1 - 2^{-k})$. \hfill $\lozenge$
Remark 9. For highest efficiency, the parameter $k$ should be chosen (see page 11 of [4]) to be the smallest integer such that

$$e \leq \frac{k(k + 1)2^{2k}}{2^{k+1} - k - 2} + 1.$$ 

Remark 10. Moreover, if a larger $e$ is desirable, $e$ should be chosen divisible by $k$. Indeed, if $e = \ell k + 1$, $e' = k(l + 1)$, $e'' = e' + 1$, then Theorem 7 yields:

$$C_A(e') = C_A(e), \quad C_D(e') = C_D(e) + 1 - 2^{1-k};$$
$$C_A(e'') = C_A(e) + 1 - 2^{-k}, \quad C_D(e'') = C_D(e) + k.$$ 

3 Sliding windows

We can further improve the $2^k$-ary algorithm if we notice that an addition is performed each time a window of length $k$ contains a non-zero bit; thus, if we allow ourselves more freedom in positioning these windows (we let them “slide”), we might end up with fewer additions, especially if $k$ is small. For example, if $k = 3$ and $n = 234 = 11.101.010_2$, Algorithm 4 requires two additions, but if we subdivide $n = 111.0101.012_2$, just one addition is required.

Before we introduce the algorithm, we explicitly state how we want to subdivide the binary digits of $n$ (and we suppose that the time to do this is negligible):

Lemma 11. Fix $k$. Every integer $n \in \mathbb{N}$ can be written in a unique way as $\sum_{i=0}^{d} \nu_i 2^{e_i}$, where $\nu_i \in \mathcal{P} = \{1, 3, 5, \ldots, 2^k - 1\}$, $e - 1 \geq e_0 > e_1 > \cdots > e_d \geq 0$ and $e_i - e_{i+1} \geq k$.

Algorithm 12.
1. while $i = 3, 5, \ldots, 2^k - 1$: store $iP$
2. let $n = \sum_{i=0}^{d} \nu_i 2^{e_i}$, $\varepsilon_{d+1} = 0$ // where the $\nu_i$ and $\varepsilon_i$ are as in Lemma 11
3. let $Q = \nu_0 P$; let $Q = 2^{e_0 - \varepsilon_1} Q$
4. while $i = 1 \ldots d$:
5. let $Q = Q + \nu_i P$
6. let $Q = 2^{e_i - \varepsilon_{i+1}} Q$
7. return $Q$

The algorithm is valid since it is clear that its output is $\sum_{i=0}^{d} \nu_i 2^{e_i}$. We have $c_A(n) = d$, while $c_D(n) = (\varepsilon_0 - \varepsilon_1) + (\varepsilon_1 - \varepsilon_2) + \cdots + (\varepsilon_d - 0) = \varepsilon_0$.

Lemma 13. Let $d(n)$, resp. $\varepsilon_0(n)$, be the value of $d$, resp. $\varepsilon_0$, as in Lemma 11; let $\delta_e = \sum_{i=1}^{d} d(n)$ and $\eta_e = \sum_{\lambda(n) = e-1} \varepsilon_0(n)$. Then $\delta_e = 0$ for $e \leq k$, while if $e > k$,

$$\delta_e = \delta_{e-1} + 2^{k-1} \delta_{e-k} + 2^{e-1} - 2^{k-1}, \quad \eta_e = 2^{e-1}(e - k + 1) - 2^{e-k}.$$ 

Proof. Using the notation of Lemma 11, let $n_0 = \nu_0 2^{e_0+k-e}$. Then $n = n_0 2^{e-k} + n'$ where $2^{k-1} \leq n_0 < 2^k$ and $0 \leq n' < 2^{e-k}$—this corresponds to taking the first window of length
which gives the first part. Now, since \( \nu_0 \) is odd, \( v_2(n_0) = \varepsilon_0 + k - e \). Thus, if \( e > k \),
\[
\eta_e = \sum_{\lambda(n_0)=k-1}^{2e-k-1} \sum_{n'=0}^{2e-k-1} \varepsilon_0(n_02^{e-k} + n') = \sum_{\lambda(n_0)=k-1}^{2e-k-1} v_2(n_0) + e - k
\]
\[
= 2^{e-k}2^{k-1}(e-k) + 2^{e-k} \sum_{\lambda(n_0)=k-1}^{2e-k} v_2(n_0),
\]
which, by Lemma 6, gives \( \eta_e = 2^{e-k}(e-k+1) - 2^{e-k} \).

**Proposition 14.** Let \( F(z) = \sum_{e=0}^{\infty} \delta_e z^e \) be the generating function of \( \delta_e \). Then
\[
F(z) = \frac{2^{k-1}z^{k+1}}{(1-2z)^2(1-z)p(z)}
\]
where \( p(x) \in \mathbb{Z}[z] \) has degree \( k-1 \), all roots have norm \( > 1/2 \), there is one real root if \( k \)
is even (in this case the root is \( < -1/2 \)) and none if \( k \) is odd.

**Proof.** By the previous lemma,
\[
F(z) = \sum_{e=0}^{\infty} \delta_e z^e + 2^{k-1} \sum_{e=0}^{\infty} \delta_{e-k} z^e + \sum_{e=k}^{\infty} 2^{e-1}z^e - 2^{k-1} \sum_{e=k}^{\infty} z^e
\]
\[
= \left( z + 2^{k-1}z^k \right) F(z) + \frac{2^{k-1}z^{k+1}}{(1-2z)(1-z)}.
\]
Thus,
\[
F(z) = \frac{2^{k-1}z^{k+1}}{(1-2z)(1-z)(1-z - 2^{k-1}z^k)}.
\]
We have \( 1 - z - 2^{k-1}z^k = (1 - 2z) \left( 1 + z + 2z^2 + \cdots + 2^{k-2}z^{k-1} \right) \); let \( q(z) \) be the LHS
and \( p(z) = q(z)/(1-2z) \). We leave it to the reader to verify that \( 1/2 \) is the only rational root of \( q(z) \). Since \( q'(z) = -1 - k(2z)^{k-1} \), the stationary points of \( q(z) \) are \( \zeta^j \xi \), where \( \xi = \exp(2\pi i/(k-1)) \), \( \xi = 1/2 \sqrt{-k} \) and \( j = 0, \ldots, k-2 \). Now, \( q(\zeta^j \xi) = 1 + \zeta^j \xi(k^{-1} - 1) \neq 0 \), since otherwise \( \zeta^j \xi \) would be a rational root. This shows that all roots of \( q(z) \) are simple.

Moreover, there is one real stationary point if \( k \) is even, none if \( k \) is odd; hence, remarking that \( q(-1/2) = 1 \), we deduce that in the former case \( q(z) \) has exactly one other real root (which is then \( < -1/2 \)) and that it has none in the latter case.

At last, suppose \( z \in \mathbb{C}, |z| \leq 1/2 \) and \( z \neq 1/2 \). Then \( |z-1| > 1/2 \) and \( |2^{k-1}z^k| = 2^{k-1}|z|^k \leq 1/2 < |z-1| \); hence \( |q(z)| > 0 \).
Theorem 15. With respect to Algorithm 12:

1. $C_A(\mathcal{P}) = 2^{k-1} - 1$, $C_D(\mathcal{P}) = 1$. If $e > k$,
   
   \[
   C_A^0(e) = \left( \frac{e}{k+1} - \frac{k^2 + 5k + 2}{2(k+1)^2} \right) + o(1), \quad C_D^0(e) = e - k + 1 - 2^{1-k}. \]

2. Let $p(z)$ be as in Proposition 14. If $k$ is even, call $\rho_0 < 0$ the real root of $p(1/2z)$ (otherwise let $\rho_0 = 0$). Let $\rho_i, \bar{\rho}_i$ be the complex roots of $p(1/2z)$, where $j = 1, \ldots, s = [(k-1)/2]$. Define $\alpha_j, \psi_j, \theta_j$ such that
   
   \[
   \alpha_j e^{\psi_j} = -\frac{1}{\rho_j^2 Q'((1/2)\rho_j)} \text{,} \quad \rho_j = \rho_j \text{,} \quad \text{if } j > 0; \quad \alpha_0 = \frac{1}{2\rho_0 Q'((1/2)\rho_0)} \text{,} \quad \text{if } k \text{ even;}
   \]
   
   where $Q(z) = (1-2z)^2(1-z)p(z)$. Then the error term for $C_A^0(e)$ is
   \[
   \alpha_0 \rho_0^2 + \sum_{j=1}^s \alpha_i \cos(\psi_j + \theta_j) |\rho_j|^e + \frac{1}{2e}. \]

Proof. Thanks to Lemma 13 and Proposition 14, the only thing we need to do is to find the power series development of $F(z)$. We will suppose $k$ even; if not, just forget about the real root $\rho_0$. Define $p^R(z) = z^{-k-1}p(1/z) = z^{-k-1} + z^{-k-2} + 2z^{-k-3} + \cdots + 2^{k-2}$ and let $\sigma_1, \ldots, \sigma_{k-1}$ be its roots (so $\sigma_i/2i = \{\rho_i, \bar{\rho}_j\}_j$). We have, by Theorem 7.30 of [7] and Proposition 14, $\delta_e = (Ae + B)2^e + C + \sum_{j=1}^s D_j \sigma_j^e$. Thus, after rearranging, we get
   \[
   \delta_e = (Ae + B)2^e + C + D_0 \cdot (2\rho_0)^e + \sum_{j=1}^s 2 \text{Re}(D_j \cdot (2\rho_j)^e), \tag{1}
   \]
   
   where $A = \frac{2}{Q''((1/2))} = \frac{1}{k+1}$, $C = -\frac{2^{k-1}}{Q'(1)} = 1$, $D_j = -\frac{1}{2\rho_j^2 Q'((1/2)\rho_j)}$. In order to compute $B$, recall that $\delta_e = 0$ for $e = 0, \ldots, k-1$. Thus,
   \[
   \begin{pmatrix}
   2^0 & \sigma_0^1 & \ldots & \sigma_0^{k-1} \\
   2^1 & \sigma_1^1 & \ldots & \sigma_1^{k-1} \\
   \vdots & \vdots & \ddots & \vdots \\
   2^{k-1} & \sigma_{k-1}^1 & \ldots & \sigma_{k-1}^{k-1}
   \end{pmatrix}
   \begin{pmatrix}
   B \\
   D_1 \\
   \vdots \\
   D_{k-1}
   \end{pmatrix}
   = -\frac{1}{k+1} \begin{pmatrix}
   k+1 \\
   k+1 - 1 \cdot 2^1 \\
   \vdots \\
   k+1 - (k-1) \cdot 2^{k-1}
   \end{pmatrix}. \tag{2}
   \]
   
   On the LHS there is a Vandermonde matrix, whose inverse $M = (m_{ij})$ is well known (see Eq. 1.2.3.40 of [8]): if we denote $[z^e]f(z)$ the coefficient of $z^e$ in $f(z)$, then
   \[
   m_{ij} = \frac{[z^{j-1}] \prod_{i=1}^{k-1} (z - \sigma_i)}{\prod_{i=1}^{k-1} (2 - \sigma_i)} = \frac{[z^{j-1}]p^R(z)}{p^R(2)} = \begin{cases}
   2^{1-j}/(k+1) & \text{if } j = 1, \ldots, k-1, \\
   2^{-j}/(k+1) & \text{if } j = k.
   \end{cases}
   \]
   
   Equation (2), hence, gives $B = -\frac{1}{k+1} \sum_{j=1}^k m_{ij} \left( k+1 + (j-1)2^{j-1} \right) = \frac{k^2 + 5k + 2}{2(k+1)^2}$. Since $|\sigma_i| < 2$ by Proposition 14, the first part of the theorem now follows. The second part is just Eq. (1) expressed in polar coordinates. \qed
Remark 16. Suppose additions and doublings have the same cost. Then the total cost, as a function of $k$, is $C(k) \sim 2^{k-1} + e/(k+1) + e - k$. Since $C(k+1) - C(k) = 2^{k-1} - e/(k^2 + 3k + 2) - 1$, the parameter $k$ should satisfy $(2^{k-1} - 1)(k^2 + 3k + 2) \leq e$, and should be chosen as the largest such integer if we are interested in maximum performance.

In particular, $k$ should not be larger that two if $e \leq 12$, three if $e \leq 60$, four if $e \leq 210$, five if $e \leq 630$, six if $e \leq 1736$.

Remark 17. For every $k$, let $\rho(k)$ be the largest in norm between the $\rho_j$. We have

<table>
<thead>
<tr>
<th>$k$</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
</tr>
</thead>
<tbody>
<tr>
<td>$</td>
<td>\rho(k)</td>
<td>$</td>
<td>$\frac{1}{2}$</td>
<td>0.707</td>
<td>0.822</td>
</tr>
<tr>
<td>$\alpha_{\rho(k)}$</td>
<td>0.091</td>
<td>0.047</td>
<td>0.014</td>
<td>0.005</td>
<td>0.002</td>
</tr>
</tbody>
</table>

It follows that it could be slightly more convenient to choose $e$ odd if $k = 2$ and, if $k = 3$, to choose $e$ such that $\cos(\psi_3 + e \theta_3) \sim 1$. For larger $k$ the error is completely negligible, at least if the costs of precomputations force $e$ to be large.

Acknowledgements
I wish to thank Roberto Avanzi the fruitful discussions on addition chains, Wenchang Chu and Alessandro Zaccagnini for telling me about generating functions.

References


