The Karatsuba Square Root Algorithm

Manuel Eberl

October 9, 2024

Abstract

This formalisation provides an executable version of Zimmerman's "Karatsuba Square Root" algorithm, which, given an integer $n \geq 0$, computes the integer square root $\lfloor\sqrt{n}\rfloor$ and the remainder $n - \lfloor\sqrt{n}\rfloor^2$. This is the algorithm used by the GNU Multiple Precision Arithmetic Library (GMP).

Similarly to Karatsuba multiplication, the algorithm is a divide-and-conquer algorithm that works by repeatedly splitting the input number n into four parts and recursively calls itself once on an input with roughly half as many bits as n, leading to a total running time of $O(M(n))$ (where $M(n)$ is the time required to multiply two n-bit numbers). This is significantly faster than the standard Heron method for large numbers (i.e. more than roughly 1000 bits).

As a simple application to interval arithmetic, an executable floating-point interval extension of the square-root operation is provided. For high-precision computations this is considerably more efficient than the interval extension method in the Isabelle distribution.

Contents

1 The function $\lceil \log_2 n \rceil$

theory *Ceil_Log2*
imports *Complex_Main*
begin

definition *ceillog2 :: "nat ⇒ nat"* **where**
"ceillog2 n = (if n = 0 then 0 else nat ⌈log 2 (real n)⌉)"

lemma *ceillog2_0 [simp]: "ceillog2 0 = 0"*
and *ceillog2_Suc_0 [simp]: "ceillog2 (Suc 0) = 0"*
and *ceillog2_2 [simp]: "ceillog2 2 = 1"*
⟨*proof*⟩

lemma *ceillog2_2_power [simp]: "ceillog2 (2 ^ n) = n"*
⟨*proof*⟩

lemma *ceillog2_ge_log:*
assumes *"n > 0"*
shows *"real (ceillog2 n) ≥ log 2 (real n)"*
⟨*proof*⟩

lemma *ceillog2_less_log:*
assumes *"n > 0"*
shows *"real (ceillog2 n) < log 2 (real n) + 1"*
⟨*proof*⟩

lemma *ceillog2_le_iff:*
assumes *"n > 0"*
shows *"ceillog2 n ≤ l ⟷ n ≤ 2 ^ l"*
⟨*proof*⟩

lemma *ceillog2_ge_iff:*
assumes *"n > 0"*
shows *"ceillog2 n ≥ l ⟷ 2 ^ l < 2 * n"*
⟨*proof*⟩

lemma *le_two_power_ceillog2: "n ≤ 2 ^ ceillog2 n"*
⟨*proof*⟩

lemma *two_power_ceillog2_gt:*
assumes *"n > 0"*
shows *"2 * n > 2 ^ ceillog2 n"*
⟨*proof*⟩

lemma *ceillog2_eqI:*
assumes *"n ≤ 2 ^ l" "2 ^ l < 2 * n"*
shows *"ceillog2 n = l"*
⟨*proof*⟩

lemma *ceillog2_rec_even:*
  assumes *"k > 0"*
  shows *"ceillog2 (2 * k) = Suc (ceillog2 k)"*
  ⟨*proof*⟩

lemma *ceillog2_mono:*
  assumes *"m ≤ n"*
  shows *"ceillog2 m ≤ ceillog2 n"*
⟨*proof*⟩

lemma *ceillog2_rec_odd:*
  assumes *"k > 0"*
  shows *"ceillog2 (Suc (2 * k)) = Suc (ceillog2 (Suc k))"*
⟨*proof*⟩

lemma *ceillog2_rec:*
  "ceillog2 n = (if n ≤ 1 then 0 else 1 + ceillog2 ((n + 1) div 2))"
⟨*proof*⟩

lemmas *[code] = ceillog2_rec*

end

1.1 Auxiliary material

theory *Karatsuba_Sqrt_Library*
imports
  Complex_Main
  "HOL-Library.Discrete"
  "HOL-Library.Log_Nat"
begin

1.2 Efficient simultaneous computation of *div* and *mod*

definition *divmod_int :: "int ⇒ int ⇒ int × int"* **where**
  "divmod_int a b = (a div b, a mod b)"

lemma *divmod_int_code [code]:*
  "divmod_int a b =
     (case divmod_integer (integer_of_int a) (integer_of_int b) of
       (q, r) ⇒ (int_of_integer q, int_of_integer r))"
  ⟨*proof*⟩

1.2.1 Missing lemmas about *bitlen*

lemma *drop_bit_eq_0_iff_nat:*
  "drop_bit k (n :: nat) = 0 ⟷ bitlen n ≤ k"
  ⟨*proof*⟩

lemma *drop_bit_eq_0_iff_int:*
assumes *"n ≥ 0"*
shows *"drop_bit k (n :: int) = 0 ⟷ bitlen n ≤ k"*
⟨*proof*⟩

lemma *drop_bit_bitlen_minus_1:*
assumes *"n > 0"*
shows *"drop_bit (nat (bitlen n - 1)) n = 1"*
⟨*proof*⟩

lemma *bitlen_pos: "n > 0 ⟹ bitlen n > 0"*
⟨*proof*⟩

lemma *bit_bitlen_minus_1:*
assumes *"n > 0"*
shows *"bit n (nat (bitlen n - 1))"*
⟨*proof*⟩

lemma *not_bit_ge_bitlen:*
assumes *"int k ≥ bitlen n" "n ≥ 0"*
shows *"¬bit n k"*
⟨*proof*⟩

lemma *bitlen_eqI:*
assumes *"bit n (nat k - 1)" "⋀i. int i ≥ k ⟹ ¬bit n i" "k > 0"*
"n ≥ 0"
shows *"bitlen n = k"*
⟨*proof*⟩

lemma *bitlen_drop_bit:*
assumes *"n ≥ 0"*
shows *"bitlen (drop_bit k n) = max 0 (bitlen n - k)"*
⟨*proof*⟩

1.2.2 Missing lemmas about *Discrete.sqrt*

lemma *Discrete_sqrt_lessI:*
assumes *"x < y ^ 2"*
shows *"Discrete.sqrt x < y"*
⟨*proof*⟩

lemma *Discrete_sqrt_conv_floor_sqrt:*
"Discrete.sqrt n = nat (floor (sqrt n))"
⟨*proof*⟩

1.3 Miscellaneous

lemma *Let_cong:*
assumes *"a = c" "⋀x. x = a ⟹ b x = d x"*

shows *"Let a b = Let c d"*
⟨*proof*⟩

lemma *case_prod_cong:*
assumes *"a = b"* *"⋀x y. a = (x, y) ⟹ f x y = g x y"*
shows *"(case a of (x, y) ⇒ f x y) = (case b of (x, y) ⇒ g x y)"*
⟨*proof*⟩

end
theory *Karatsuba_Sqrt*
imports
Complex_Main
"HOL-Library.Discrete"
"HOL-Library.Log_Nat"
Ceil_Log2
Karatsuba_Sqrt_Library
begin

1.4 Definition of an integer square root with remainder

definition *sqrt_rem :: "nat ⇒ nat"* **where**
"sqrt_rem n = n - Discrete.sqrt n ^ 2"

lemma *sqrt_rem_upper_bound: "sqrt_rem n ≤ 2 * Discrete.sqrt n"*
⟨*proof*⟩

lemma *of_nat_sqrt_rem:*
"(of_nat (sqrt_rem n) :: 'a :: ring_1) = of_nat n - of_nat (Discrete.sqrt n) ^ 2"
⟨*proof*⟩

definition *sqrt_rem'* **where** *"sqrt_rem' n = (Discrete.sqrt n, sqrt_rem n)"*

lemma *Discrete_sqrt_code [code]: "Discrete.sqrt n = fst (sqrt_rem' n)"*
⟨*proof*⟩

lemma *sqrt_rem_code [code]: "sqrt_rem n = snd (sqrt_rem' n)"*
⟨*proof*⟩

1.5 Heron's method

The method used here is a variant of Heron's method, which is itself essentially Newton's method specialised to square roots. This is already in the AFP under the name "Babylonian method". However, that entry derives a more general version for n-th roots and lacks some flexibility that is useful for us here, so we instead derive a simple version for the square root directly. We will use this version in the base case of the algorithm.

The starting value must be bigger than $\lfloor\sqrt{n}\rfloor$. We simply use $2^{\lceil \frac{1}{2} \log_2 n \rceil}$,

which is easy to compute and fairly close to $\sqrt{n}$ already so that the Newton iterations converge very quickly.

context
  fixes *n :: nat*
begin

function *sqrt_rem_aux :: "nat ⇒ nat × nat"* **where**
  "sqrt_rem_aux x =
    (if x² ≤ n then (x, n - x²) else sqrt_rem_aux ((n div x + x) div 2))"
  ⟨*proof*⟩
termination ⟨*proof*⟩

lemmas *[simp del] = sqrt_rem_aux.simps*

lemma *sqrt_rem_aux_code [code]:*
  "sqrt_rem_aux x = (
    *let x2 = x*x; r = int n - int x2*
    in if r ≥ 0 then (x, nat r) else sqrt_rem_aux (drop_bit 1 (n div x + x)))"
  ⟨*proof*⟩

lemma *sqrt_rem_aux_decompose: "fst (sqrt_rem_aux x) ^ 2 + snd (sqrt_rem_aux x) = n"*
  ⟨*proof*⟩

lemma *sqrt_rem_aux_correct:*
  assumes *"x ≥ Discrete.sqrt n"*
  shows *"fst (sqrt_rem_aux x) = Discrete.sqrt n"*
  ⟨*proof*⟩

lemma *sqrt_rem_aux_correct':*
  assumes *"x ≥ Discrete.sqrt n"*
  shows *"sqrt_rem_aux x = sqrt_rem' n"*
  ⟨*proof*⟩

definition *sqrt_rem'_heron :: "nat × nat"* **where**
  "sqrt_rem'_heron = sqrt_rem_aux (push_bit ((ceillog2 n + 1) div 2) 1)"

lemma *sqrt_rem'_heron_correct:*
  "sqrt_rem'_heron = sqrt_rem' n"
⟨*proof*⟩

end

lemmas *[code] = sqrt_rem'_heron_correct [symmetric]*

1.6 Main algorithm

1.6.1 Single step

definition *splice_bit* **where**
```
  "splice_bit i n x = take_bit n (drop_bit i x)"
```

lemma *of_nat_splice_bit:*
```
  "of_nat (splice_bit i n x) =
     splice_bit i n (of_nat x :: 'a :: linordered_euclidean_semiring_bit_operations)"
```
⟨*proof*⟩

definition *karatsuba_sqrt_step* **where**
```
  "karatsuba_sqrt_step a32 a1 a0 b =
     (let (s, r) = sqrt_rem' a32;
          (q, u) = ((r * b + a1) div (2 * s), (r * b + a1) mod (2 * s));
          s' = int (s * b + q);
          r' = int (u * b + a0) - int (q ^ 2)
      in  if r' ≥ 0 then (s', r') else (s' - 1, r' + 2 * s' - 1))"
```

definition *karatsuba_sqrt_step' :: "nat ⇒ nat ⇒ int × int"* **where**
```
  "karatsuba_sqrt_step' n k =
     (let (s, r) = map_prod int int (sqrt_rem' (drop_bit (2*k) n));
          (q, u) = divmod_int (push_bit k r + splice_bit k k n) (push_bit
1 s);
          s' = push_bit k s + q;
          r' = push_bit k u + take_bit k n - q ^ 2
      in  if r' ≥ 0 then (s', r') else (s' - 1, r' + push_bit 1 s' - 1))"
```

Note that unlike Zimmerman, we do not have any upper bound on a_3 since this bound turned out to be unnecessary for the correctness of the algorithm. As long as b^4 is not much smaller than n, there is no efficiency problem either, since the step will still strip away about half of the bits of n.

The advantage of this is that we do not have to do the "normalisation" done by Zimmerman to ensure that at least one of the two most significant bits of a_3 be set.

lemma *karatsuba_sqrt_step_correct:*
fixes *a32 a1 a0 :: nat*
assumes *"a1 < b" "a0 < b" "4 * a32 ≥ b ^ 2" "even b"*
defines *"n ≡ a32 * b ^ 2 + a1 * b + a0"*
shows *"karatsuba_sqrt_step a32 a1 a0 b =*
map_prod of_nat of_nat (sqrt_rem' n)"
⟨*proof*⟩

lemma *karatsuba_sqrt_step'_correct:*
fixes *k n :: nat*
assumes *k: "k > 0"* **and** *bitlen: "int k ≤ (bitlen n + 1) div 4"*
defines *"a32 ≡ drop_bit (2*k) n"*
defines *"a1 ≡ splice_bit k k n"*

defines *"a0 ≡ take_bit k n"*
shows *"karatsuba_sqrt_step' n k = map_prod int int (sqrt_rem' n)"*
⟨*proof*⟩

1.6.2 Full algorithm

Our algorithm is parameterised with a "limb size" and a cutoff. The cutoff value describes the threshold for the base case, i.e. the size of inputs (in bits) for which we fall back to Heron's method.

The algorithm splits the input number into four parts in such a way that the bit size of the lower three parts is a multiple of 2^l (where l is the limb size). This may be useful to avoid unnecessary bit shifting, since one one always splits the input number exactly at limb boundaries. However, whether this actually helps depends on how bit shifting of arbitrary-precision integers is actually implemented in the runtime.

There is only one rather weak condition on the limb size and cutoff. Which values work best must be determined experimentally.

locale *karatsuba_sqrt =*
fixes *cutoff limb_size :: nat*
assumes *cutoff: "2 ^ (2 + limb_size) ≤ cutoff + 2"*
begin

function *karatsuba_sqrt_aux :: "nat ⇒ int × int"* **where**
"karatsuba_sqrt_aux n = (
let sz = bitlen n
in if sz ≤ int cutoff then
case sqrt_rem'_heron n of (s, r) ⇒ (int s, int r)
else let
k = push_bit limb_size (drop_bit (2 + limb_size) (nat (bitlen n + 1)));
*(s, r) = karatsuba_sqrt_aux (drop_bit (2*k) n);*
(q, u) = divmod_int (push_bit k r + splice_bit k k n) (push_bit 1 s);
s' = push_bit k s + q;
r' = push_bit k u + take_bit k n - q ^ 2
in if r' ≥ 0 then (s', r') else (s' - 1, r' + push_bit 1 s' - 1))"

⟨*proof*⟩
termination ⟨*proof*⟩

lemmas *[simp del] = karatsuba_sqrt_aux.simps*

lemma *karatsuba_sqrt_aux_correct: "karatsuba_sqrt_aux n = map_prod int int (sqrt_rem' n)"*
⟨*proof*⟩

definition *karatsuba_sqrt* **where**

```
  "karatsuba_sqrt n = (case karatsuba_sqrt_aux n of (s, r) ⇒ (nat s,
nat r))"

theorem karatsuba_sqrt_correct: "karatsuba_sqrt n = sqrt_rem' n"
  ⟨proof⟩

end
```

1.6.3 Concrete instantiation

We pick a cutoff of 1024 and a limb size of 64 as reasonable default values.

```
definition karatsuba_sqrt_default where
  "karatsuba_sqrt_default = karatsuba_sqrt.karatsuba_sqrt 1024 6"

definition karatsuba_sqrt_default_aux where
  "karatsuba_sqrt_default_aux = karatsuba_sqrt.karatsuba_sqrt_aux 1024
6"

interpretation karatsuba_sqrt_default:
  karatsuba_sqrt 1024 6
  rewrites "karatsuba_sqrt.karatsuba_sqrt 1024 6 ≡ karatsuba_sqrt_default"
       and "karatsuba_sqrt.karatsuba_sqrt_aux 1024 6 ≡ karatsuba_sqrt_default_aux"
  ⟨proof⟩

lemmas [code] =
  karatsuba_sqrt_default.karatsuba_sqrt_aux.simps[unfolded power2_eq_square]
  karatsuba_sqrt_default.karatsuba_sqrt_def
  karatsuba_sqrt_default.karatsuba_sqrt_correct [symmetric]
```

1.7 Using *sqrt_rem* to compute floors and ceilings of *sqrt*

```
definition sqrt_nat_ceiling :: "nat ⇒ nat" where
  "sqrt_nat_ceiling n = nat (ceiling (sqrt (real n)))"

definition sqrt_int_floor :: "int ⇒ int" where
  "sqrt_int_floor n = floor (sqrt (real_of_int n))"

definition sqrt_int_ceiling :: "int ⇒ int" where
  "sqrt_int_ceiling n = ceiling (sqrt (real_of_int n))"

lemma sqrt_nat_ceiling_code [code]:
  "sqrt_nat_ceiling n = (case sqrt_rem' n of (s, r) ⇒ if r = 0 then s
else s + 1)"
⟨proof⟩

lemma sqrt_int_floor_code [code]:
  "sqrt_int_floor n =
     (if n ≥ 0 then int (Discrete.sqrt (nat n)) else -int (sqrt_nat_ceiling
(nat (-n))))"
```

⟨*proof*⟩

lemma *sqrt_int_ceiling_code [code]:*
"sqrt_int_ceiling n =
(if n ≥ 0 then int (sqrt_nat_ceiling (nat n)) else -int (Discrete.sqrt (nat (-n))))"
⟨*proof*⟩

end
theory *Karatsuba_Sqrt_Float*
imports
Karatsuba_Sqrt
"HOL-Library.Interval_Float"
begin

1.8 Floating-point approximation of *sqrt*

definition *shift_int :: "int ⇒ int ⇒ int"*
where *"shift_int k n = (if k ≥ 0 then n * 2 ^ nat k else n div 2 ^ (nat (-k)))"*

lemma *shift_int_code [code]:*
"shift_int k n = (if k ≥ 0 then push_bit (nat k) n else drop_bit (nat (-k)) n)"
⟨*proof*⟩

definition *lb_sqrt :: "nat ⇒ float ⇒ float"* **where**
*"lb_sqrt prec x = (let n = mantissa x; e = exponent x; k = nat (2 * int prec - bitlen n);*
k' = (if even k = even e then k else k + 1) in
normfloat (Float (sqrt_int_floor (shift_int k' n)) (shift_int (-1) (e - k'))))"

definition *ub_sqrt :: "nat ⇒ float ⇒ float"* **where**
*"ub_sqrt prec x = (let n = mantissa x; e = exponent x; k = nat (2 * prec - bitlen n);*
k' = (if even k = even e then k else k + 1) in
normfloat (Float (sqrt_int_ceiling (shift_int k' n)) (shift_int (-1) (e - k'))))"

lemma *lb_sqrt: "lb_sqrt prec x ≤ sqrt x"*
⟨*proof*⟩

lemma *ub_sqrt: "ub_sqrt prec x ≥ sqrt x"*
⟨*proof*⟩

context
includes *interval.lifting*
begin

lift_definition *sqrt_float_interval :: "nat ⇒ float interval ⇒ float interval"* **is**
"λprec (l, u). (lb_sqrt prec l, ub_sqrt prec u)"
⟨*proof*⟩

lemma *sqrt_float_intervalI:*
fixes *x :: real* **and** *X :: "float interval"*
assumes *"x ∈ set_of (real_interval X)"*
shows *"sqrt x ∈ set_of (real_interval (sqrt_float_interval prec X))"*
⟨*proof*⟩

lemma *sqrt_float_interval:*
"sqrt ‘ set_of (real_interval X) ⊆ set_of (real_interval (sqrt_float_interval prec X))"
⟨*proof*⟩

end

end

1.9 Tests

theory *Karatsuba_Sqrt_Test*
imports
Karatsuba_Sqrt_Float
"HOL-Library.Code_Target_Numeral"
begin

value *"sqrt_rem' 123456"*
value *"sqrt_rem 123456"*
value *"Discrete.sqrt 123456"*
value *"sqrt_int_floor 123456"*
value *"sqrt_nat_ceiling 123456"*
value *"sqrt_int_ceiling 123456"*
value *"sqrt_float_interval 64 (Ivl 123456 123456)"*

end

References

[1] Y. Bertot, N. Magaud, and P. Zimmermann. A proof of GMP square root. *Journal of Automated Reasoning*, 29(3):225–252, 2002.

[2] P. Zimmermann. Karatsuba Square Root. Research Report RR-3805, INRIA, 1999.