UNIVERSITÄT HEIDELBERG
ZUKUNFT SEIT 1386

Energy efficient calculation of simple functions

Advanced Seminar Computer Engineering

Abdulhamid Han

19.01.2016

Energy efficiency depends also from the algorithm

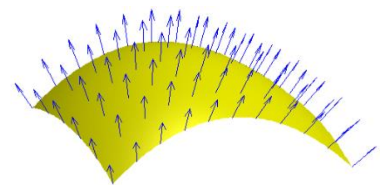
For example:

bubblesort $O(n^2) \leftrightarrow$ quicksort $O(n \cdot \log n)$

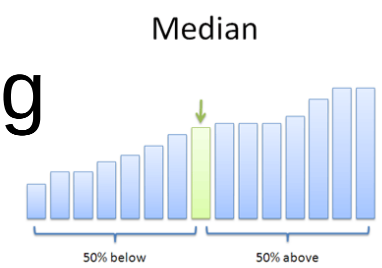
$$n = 10^6 \rightarrow relative\ deviation \approx 10^5$$

Content

- Fast inverse square root



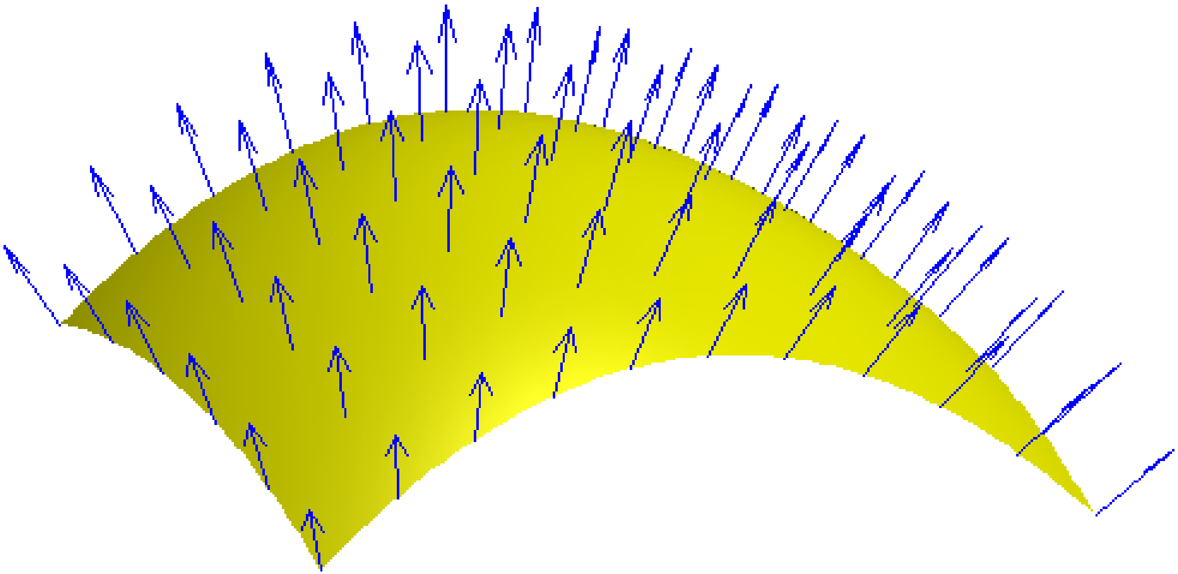
- Finding the median without sorting



- Bit counting

28 = | 1 | 1 | 1 | 0 | 0 | → 3

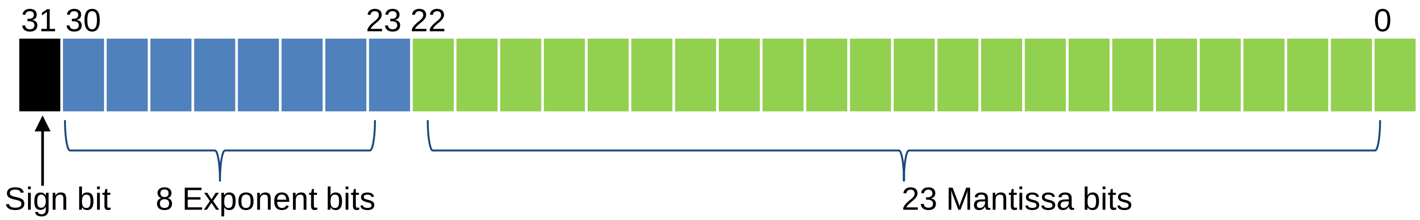
Fast inverse square root



Fast inverse square root

UNIVERSITÄT HEIDELBERG
ZUKUNFT SEIT 1386

– Single precision floating numbers are stored as 32 bit numbers



IEEE 754 Single Precision Format

$$\rightarrow x = (-1)^{\text{sign}} \cdot (1+\text{Mantissa}) \cdot 2^{\text{Exponent}-127}$$

π ≈ 0 | 1 0 0 0 0 0 0 0 | 1 0 0 1 0 0 1 0 0 0 0 1 1 1 1 1 1 0 1 1 0 1 1

$$\approx (-1)^0 \cdot (1.5707963705062866) \cdot 2^{128-127}$$

$$\approx 3.1415927$$

Fast inverse square root

UNIVERSITÄT HEIDELBERG
ZUKUNFT SEIT 1386

- In video games the inverse square root is necessary due to vector normalization
- Often the speed is more importantly than the accuracy and an accuracy of 1% is acceptable
- The main goal is to get a good approximate value in <u>one</u> calculation step

How can you calculate the inverse square without division and $\sqrt{}$?

Fast inverse square root

Integer: 26 = 1 1 0 1 0

26 >> 1 = 0 1 1 0 1 $= 13 = \frac{26}{2}$

Float: π ≈ 0 1 0 0 0 0 0 0 0 1 0 0 1 0 0 1 0 0 0 0 1 1 1 1 1 1 0 1 1 0 1 1

π >> 1 ≈ 0 0 1 0 0 0 0 0 0 0 1 0 0 1 0 0 1 0 0 0 0 1 1 1 1 1 1 0 1 1 0 1

→ $1.6263033 \cdot 10^{-19}$

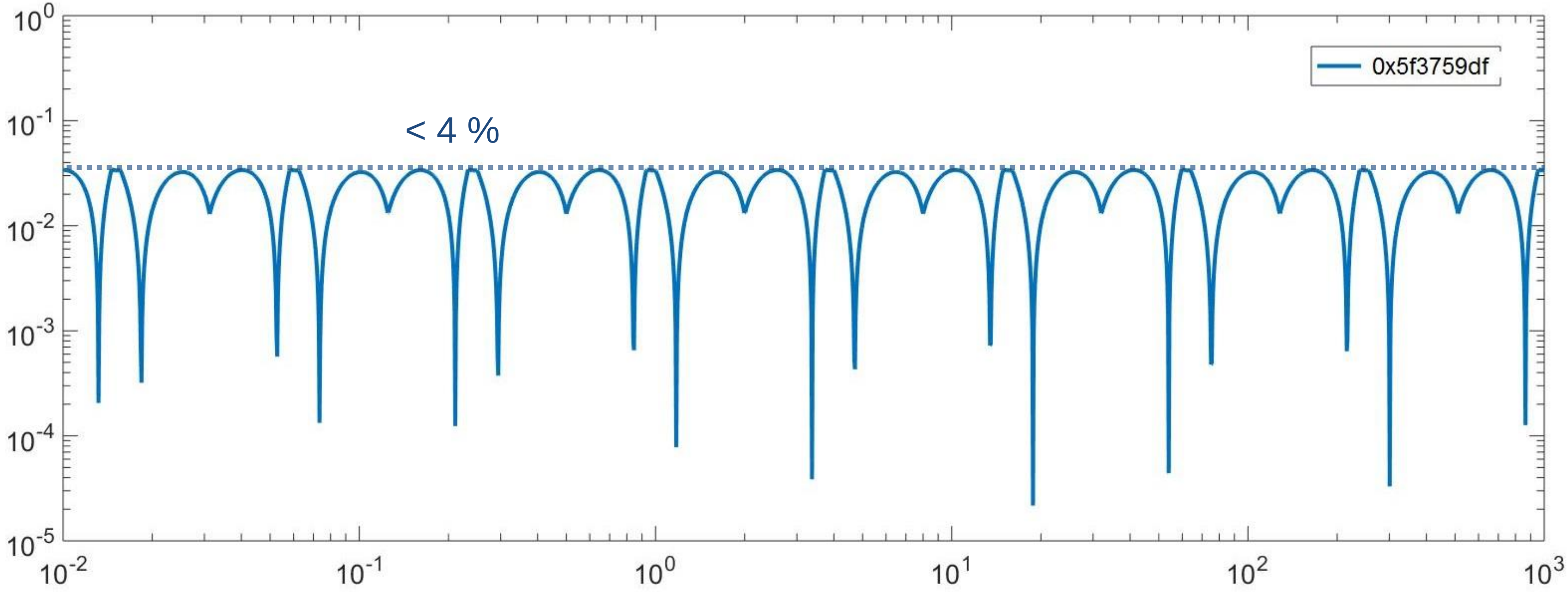
Now calculate **0x5f3759df - (π >> 1)** **(bitwise calculation!)**

→ $0.563957 \approx \frac{1}{\sqrt{\pi}}$

0 1 0 1 1 1 1 1 0 0 1 1 0 1 1 1 0 1 0 1 1 0 0 1 1 1 0 1 1 1 1 1

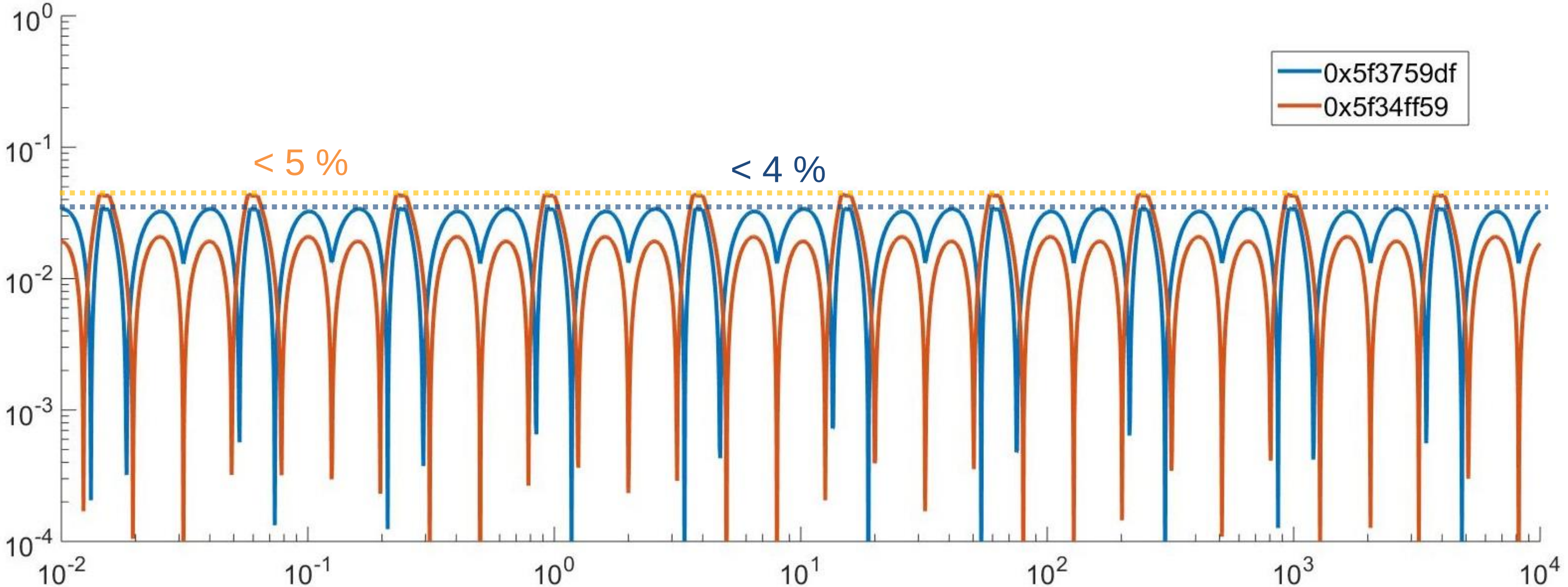
Result with 0x5f3759df

$$\frac{\frac{1}{\sqrt{x}} - InvSqrt(x)}{\frac{1}{\sqrt{x}}}$$

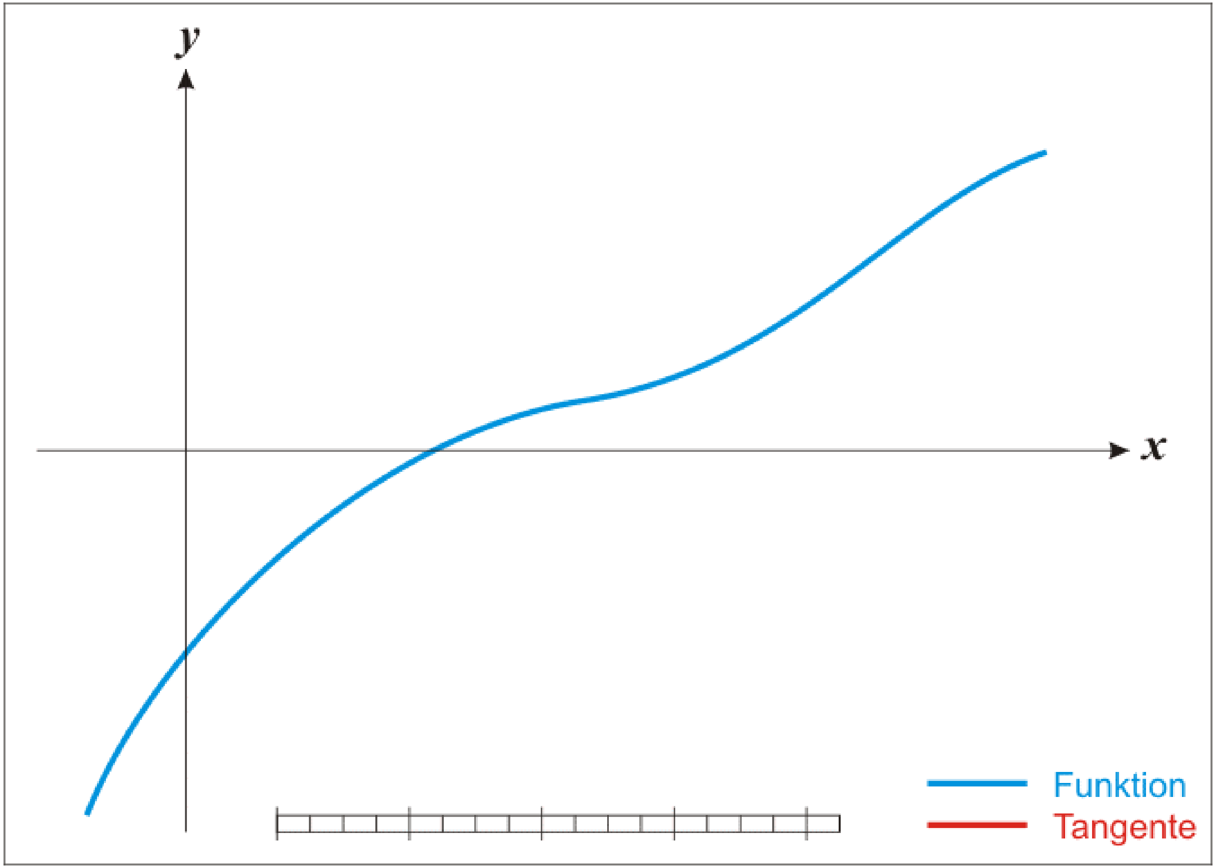


0x5f3759df vs 0x5f34ff59

$$\frac{\frac{1}{\sqrt{x}} - InvSqrt(x)}{\frac{1}{\sqrt{x}}}$$



Newton’s method



https://en.wikipedia.org/wiki/Newton%27s_method#/media/File:NewtonIteration_Ani.gif

Fast inverse square root

– An appropriate formula is $f(y) = \frac{1}{y^2} - x$

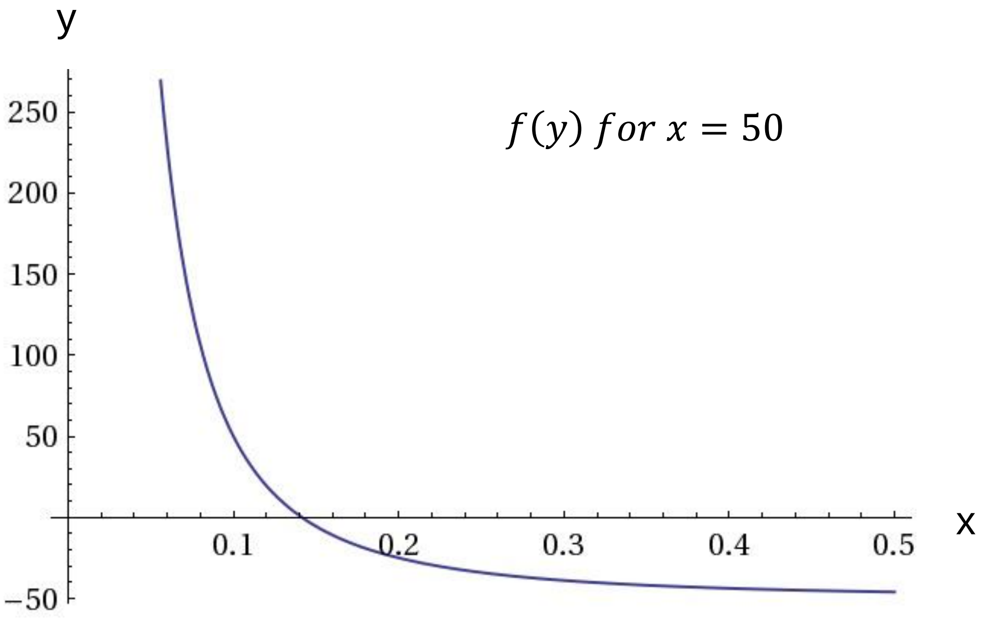
- $f(y) = 0 \rightarrow y = \frac{1}{\sqrt{x}}$

– Newton iteration

- $y_{n+1} = y_n - \frac{f(y_n)}{f\prime(y_n)}$

– Deliver

- $y_{n+1} = \frac{1}{2} y_n (3 - x y_n^2)$



Fast inverse square root

UNIVERSITÄT HEIDELBERG
ZUKUNFT SEIT 1386

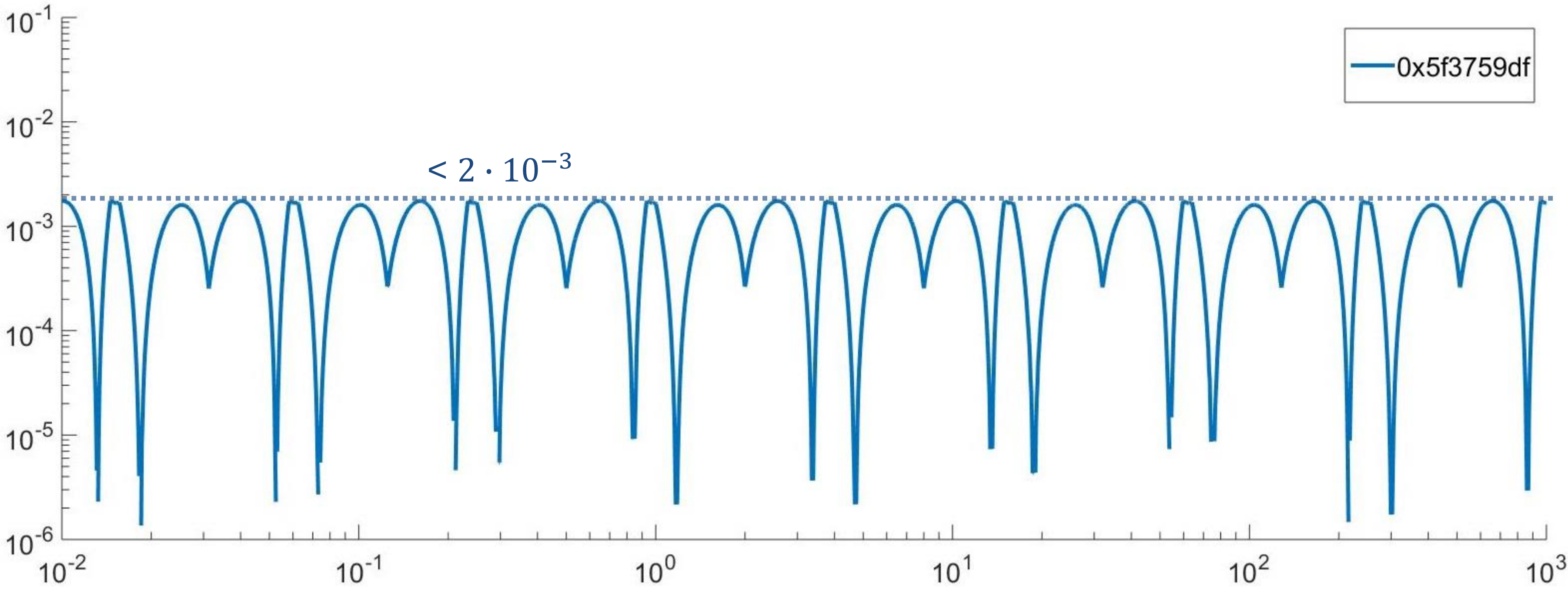
```
float InvSqrt( float number ) {
  long i;
  float x2, y;
  const float threehalfs = 1.5F;
  x2 = number * 0.5F;
  y = number;
  i = * ( long * ) &y;                        // store floating-point bits in long
  i = 0x5f3759df - ( i >> 1 );                // initial guess for Newton's method
  y = * ( float * ) &i;                       // convert new bits into float
  y = y * ( threehalfs - ( x2 * y * y ) );    // 1st iteration
  return y;
}
```

http://betterexplained.com/articles/understanding-quakes-fast-inverse-square-root/

π	≈	0	1	0	0	0	0	0	0	0	1	0	0	1	0	0	1	0	0	0	0	1	1	1	1	1	1	0	1	1	0	1	1

Result with 0x5f3759df (1 newton step)

$$\frac{\frac{1}{\sqrt{x}} - InvSqrt(x)}{\frac{1}{\sqrt{x}}}$$



Magic number for another exponents

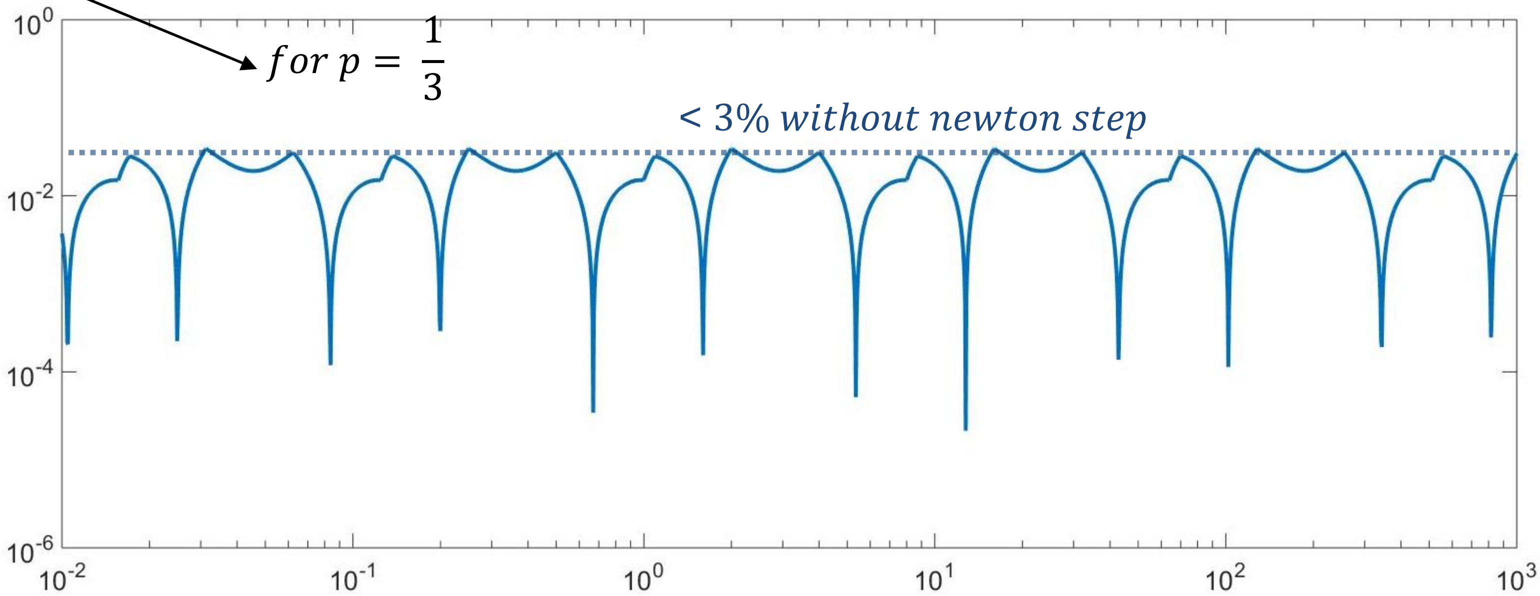
UNIVERSITÄT HEIDELBERG
ZUKUNFT SEIT 1386

Calculate $f(x) = x^p$

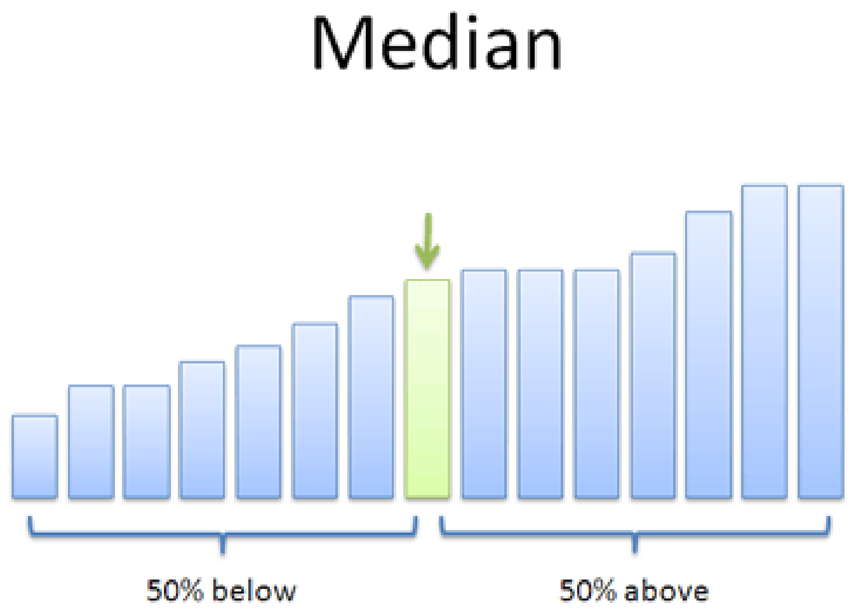
p=0.5 (square root) → i = 0x1fbd1df5 + (i >> 1)

http://h14s.p5r.org/2012/09/0x5f3759df.html

for -1 ≤ p ≤ 1 → i = (1 - p) * 0x3f7a3bea + (p * i)



Finding the median without sorting



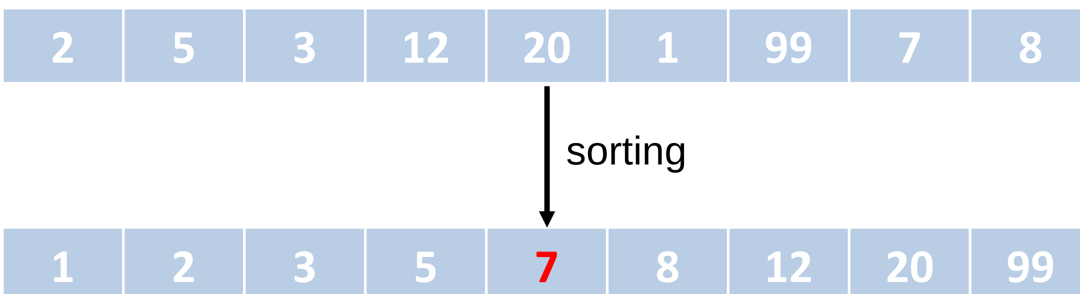
Finding the median without sorting

UNIVERSITÄT HEIDELBERG
ZUKUNFT SEIT 1386

Definition:

Median is the middle value in a sorted array

– It’s easy to find the median in a sorted array



Median?

O(n·log(n))

– In an unsorted array one can find the median **without sorting** O(n)

A simple algorithm to find the median

UNIVERSITÄT HEIDELBERG
ZUKUNFT SEIT 1386

2	5	3	12	20	1	99	7	8

- Choose an arbitrary element x

2	5	3	**12**	20	1	99	7	8

- Partition in 3 sections

2	5	3	1	7	8	**12**	20	99
a0	a1	a2	a3	a4	a5	a6	a7	a8

- Rank of median: $k = \frac{n}{2} = \frac{9}{2} = 4$
- $\rightarrow$ return

2	5	3	1	7	8

- Choose an arbitrary element x and partition in 3 sections

2	5	3	1	**7**	8
a0	a1	a2	a3	a4	a5

A simple algorithm to find the median

UNIVERSITÄT HEIDELBERG ZUKUNFT SEIT 1386

Input: array $a_0, a_1, \dots, a_{n-1}$ with length n
Output: median = element with rank m = $\frac{n}{2}$

1. If n=1 return a_0
   else
2. Choose an arbitrary element x
3. Partition the array in three sections
   1. $a_0, \dots, a_{q-1}$ with elements less than x
   2. $a_q, \dots, a_{g-1}$ with elements equal x
   3. $a_g, \dots, a_{n-1}$ with elements greater than x
4. If $m < q$ return a_m in first section
   If $m < g$ return x
   else return a_m in third section

Best case: 3 sections of equal length → $O(n)$
Worst case: returned section is always smaller by 1 → $O(n^2)$

A simple algorithm to find the median

Worst Case:

→ x should be select carefully !

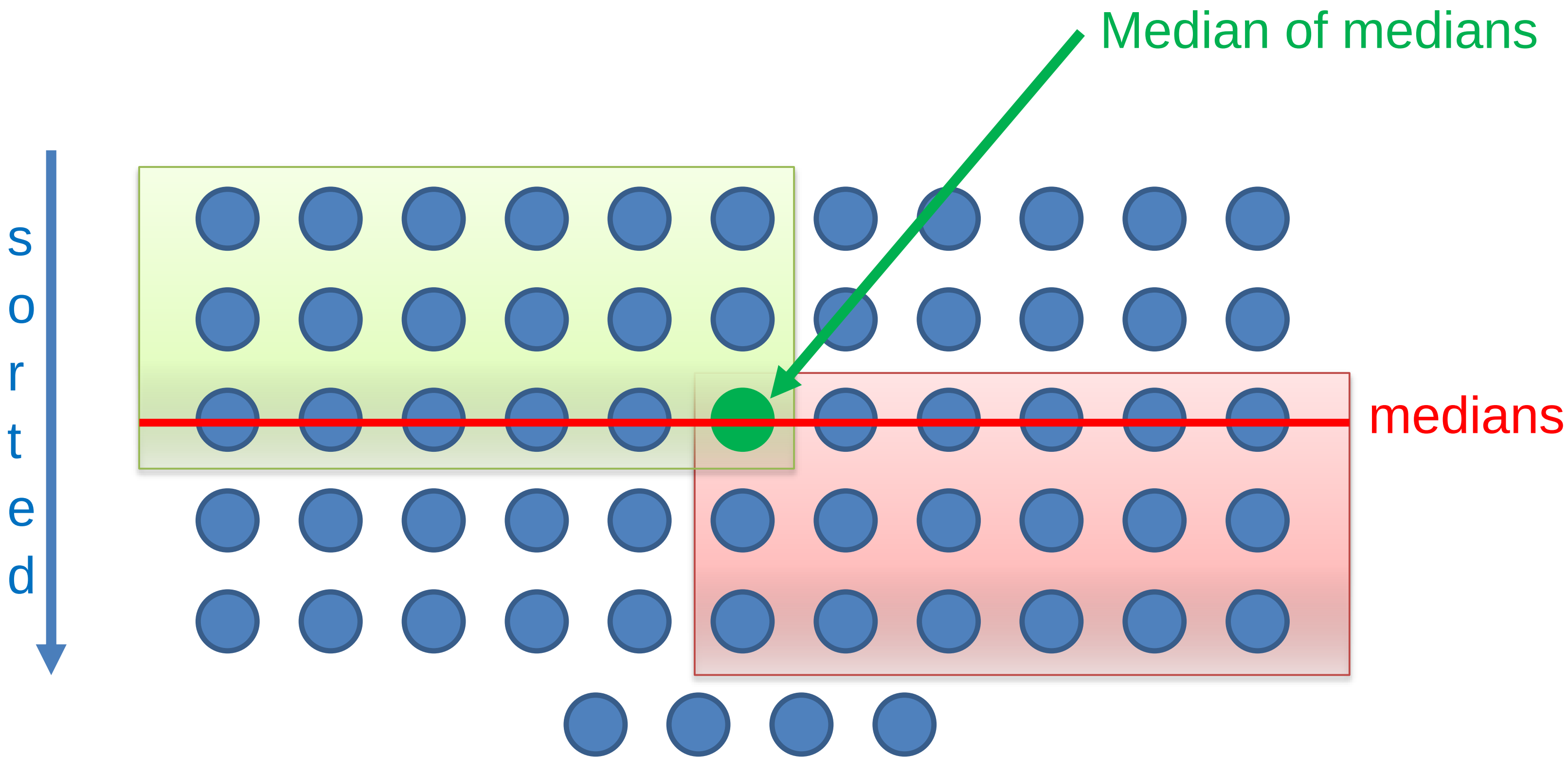
Improved version

UNIVERSITÄT HEIDELBERG
ZUKUNFT SEIT 1386

Input: array $a_0, a_1, \ldots, a_{n-1}$ with length n
Output: median = element with rank $m = \frac{n}{2}$

1. **If n<15 sort the array and return median**
   else
2. **Partition the array in $\frac{n}{5}$ sections with 5 elements and calculate their median**
3. **Calculate recursively the median m' of this medians**
4. Partition the array in three sections
   1. $a_0, \ldots, a_{q-1}$ with elements **less than m'**
   2. $a_q, \ldots, a_{g-1}$ with elements **equal m'**
   3. $a_g, \ldots, a_{n-1}$ with elements **greater than m'**
5. If $m < q$ return $a_{m'}$ in first section
   If $m < g$ **return m'**
   else return $a_{m'}$ in third section

Improved version



Up to 4 additional elements, if n is not divisible by 5

Improved version

4	62	100	5	66
33	5	342	14	3
22	1	14	124	55
7	52	78	51	45
42	26	24	79	82

→

4	1	14	5	3
7	5	24	14	45
22	26	78	51	55
33	52	100	79	66
42	62	342	124	82

↓

< 51

4	1	14	5	3
7	5	24	14	45
22	26	**51**	55	78
33	52	100	79	66
42	62	342	124	82

> 51

Improved version

UNIVERSITÄT HEIDELBERG
ZUKUNFT SEIT 1386

Input: array $a_0, a_1, \dots, a_{n-1}$ with length n
Output: median = element with rank m = $\frac{n}{2}$

1. **If n<15 sort the array and return median** O(1)
   else
2. **Partition the array in $\frac{n}{5}$ sections with 5 elements and calculate their median** O(n)
3. **Calculate recursively the median m' of this medians** O(n)
4. Partition the array in three sections O(n)
   1. $a_0, \dots, a_{q-1}$ with elements **less than m'**
   2. $a_q, \dots, a_{g-1}$ with elements **equal m'**
   3. $a_g, \dots, a_{n-1}$ with elements **greater than m'**
5. If m < q return $a_{m'}$ in first section
   If m < g **return m** ≤T(3n/4)
   else return $a_{m'}$ in third section

Bit Counting

28 =	1	1	1	0	0	→ 3

Bit counting

UNIVERSITÄT HEIDELBERG ZUKUNFT SEIT 1386

Simple Solution

```
unsigned int c = 0;
 for (unsigned int mask = 0x1; mask; mask<<=1) {  // 32 loops! Repeat until
                                                     mask == 0
   if (v & mask) c++;
 }
```

Disadvantage: always 32 loops

Bit counting

UNIVERSITÄT HEIDELBERG
ZUKUNFT SEIT 1386

First improvement

```
unsigned int c;
for (c = 0; v; v >>= 1) {                              // shift while v!=0
  c+= v & 1;                                           // increase counter
}
```

Disadvantage: as many loops as the highest set bit

v=0x1 → 1 loop

v=0x80000000 → 32 loop

Bit counting

UNIVERSITÄT HEIDELBERG ZUKUNFT SEIT 1386

Second improvement

```
unsigned int c;
for (c = 0; v; c++) {                          // repeat until v == 0
 v &= v - 1;                                   // delete lowest set bit
}
```

v = …xyz10…0

v-1 = …xyz01…1

→ v & v-1 = …xyz0…0

Advantage: as many loops as the number of ones

But still not fast enough if the number of ones is large

An elegant method

UNIVERSITÄT HEIDELBERG
ZUKUNFT SEIT 1386

v = ab | ab | … | ab | ab   (16 times 2 bits)

c – number of ones

a	b	c	ab - 0a
0	0	00	00
0	1	01	01
1	0	01	01
1	1	10	10

ab – 0a can be calculated with `v - ((v >> 1) & 0x55555555)`

An elegant method

Now add 2 neighbor 2 bits to a 4 bit

v = ab* | ab* | … | ab* | ab*   (16 times 2 bits)

v = ab’+ab” | … | ab’+ab”   (8 times 4 bit)

- No carry!

It can be calculated with:

```
(v & 0x33333333) + ((v >> 2) & 0x33333333);
```

An elegant method

UNIVERSITÄT HEIDELBERG
ZUKUNFT SEIT 1386

Now sum up 2 neighbor 4 bits to a 8 bit:

```
1.  v = (v + (v >> 4));
2.  v &= 0x0F0F0F0F;        // delete useless bits
```

- Still no carry !

v contains 4 times 8 bit (v=ABCD)

v*0x01010101 = **D**000 + **C**D00 + **B**CD0 + **A**BCD

>> 24 deliver A+B+C+D

The result is:

```
c = (v * 0x01010101) >> 24;
```

An elegant method

UNIVERSITÄT HEIDELBERG
ZUKUNFT SEIT 1386

```
v = v - ((v >> 1) & 0x55555555);                   // count bits in two groups
v = (v & 0x33333333) + ((v >> 2) & 0x33333333);   // Add 2 groups-> 4 groups
v = (v + (v >> 4));                                // Add 4 groups-> 8 groups
v &= 0x0F0F0F0F;                                  // delete useless bits
c = (v * 0x01010101) >> 24;                        // Add the 4 8 groups
```

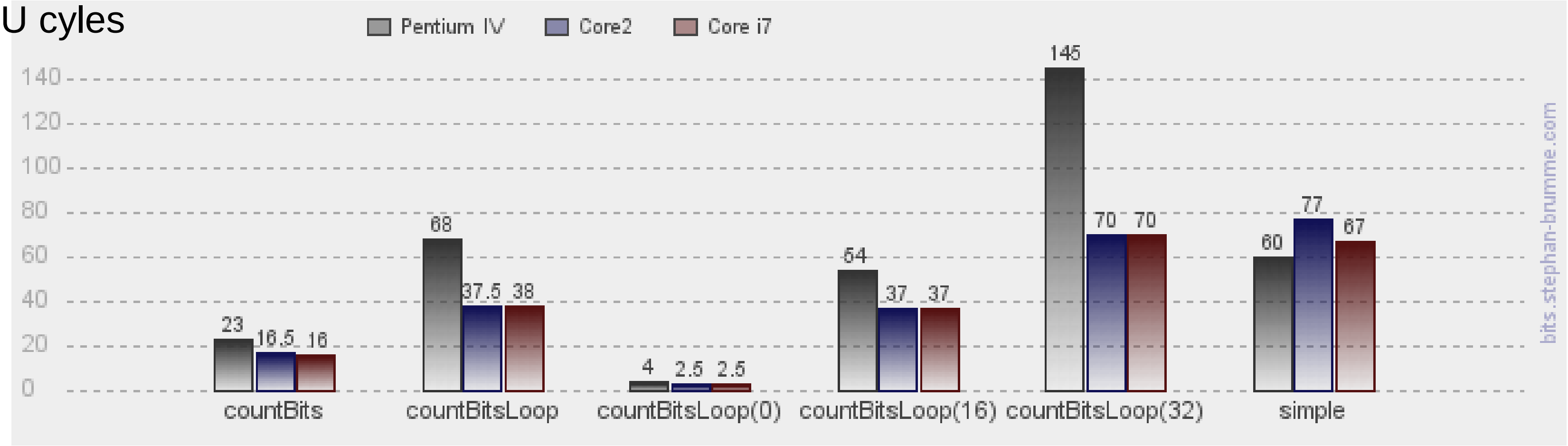
```
Advantage:       count bits in constant time
Disadvantage:    not optimal in a few bits set
```

Results

UNIVERSITÄT
HEIDELBERG
ZUKUNFT
SEIT 1386

Second improvement vs. elegant method

CPU cyles



http://bits.stephan-brumme.com/countBits.html

Conclusion

- Fast inverse square root
  - One can calculate the inverse square root 4 times faster with an accuracy of < 1%
- Finding the median without sorting
  - One can find the median without sorting
  - The complexity is O(n)
- Bit counting
  - It’s possible to count set bits in constant time independent of the input value