# Good Predictions Are Worth a Few Comparisons

Carine Pivoteau

with Nicolas Auger and Cyril Nicaud

LIGM - Université Paris-Est-Marne-la-Vallée

March 2016

# A case study

Find both the min. and the max. of an array of size $n$.

**Naive Algorithm:**

| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| 5 | 1 | 4 | 3 | 6 | 0 | 2 | 8 | 7 | 9 |

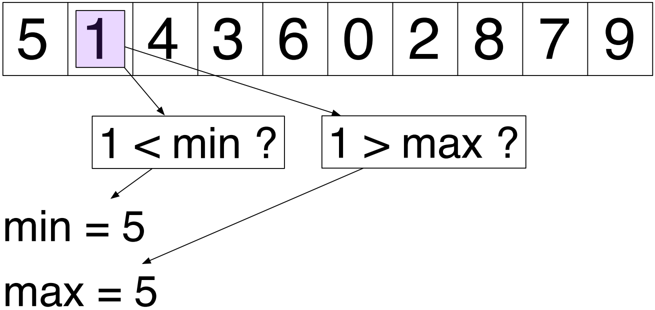
min = 5

max = 5

# A case study

Find both the min. and the max. of an array of size $n$.

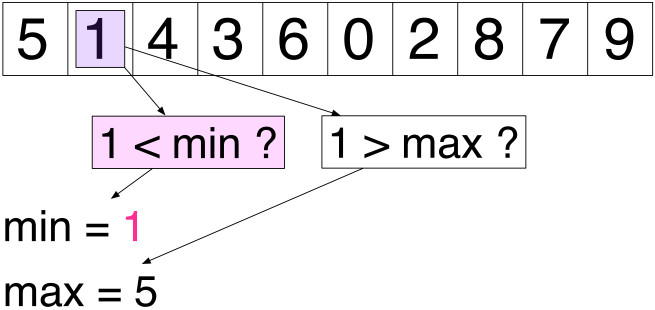
**Naive Algorithm:**



# A case study

Find both the min. and the max. of an array of size $n$.

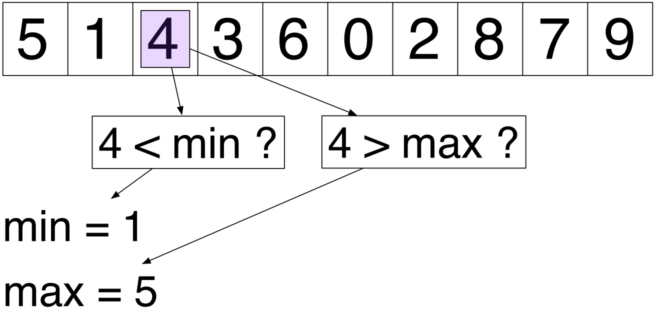
**Naive Algorithm:**



# A case study

Find both the min. and the max. of an array of size $n$.

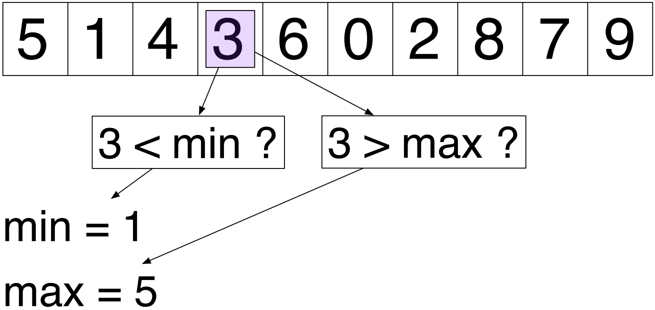
**Naive Algorithm:**



# A case study

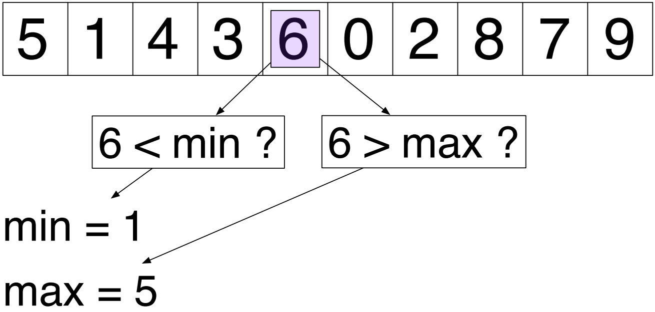
Find both the min. and the max. of an array of size $n$.

**Naive Algorithm:**



Find both the min. and the max. of an array of size $n$.

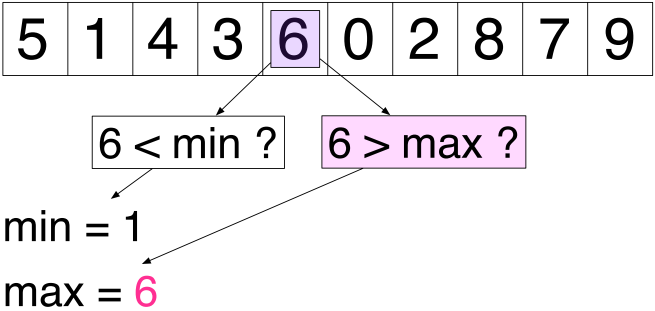
**Naive Algorithm:**



# A case study

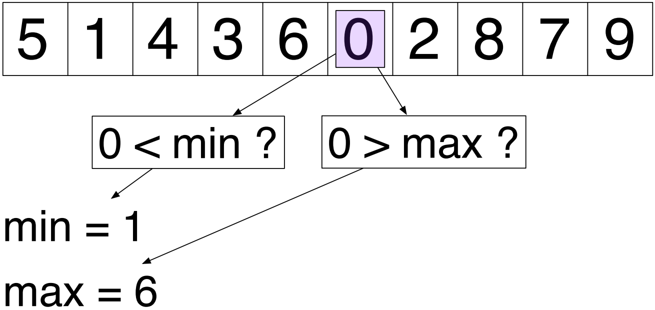
Find both the min. and the max. of an array of size $n$.

**Naive Algorithm:**



Find both the min. and the max. of an array of size $n$.

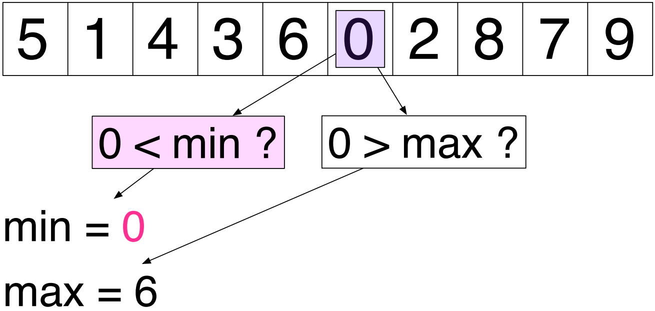
**Naive Algorithm:**



# A case study

Find both the min. and the max. of an array of size $n$.

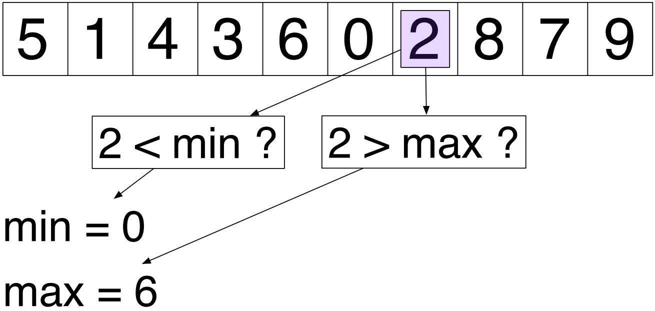
**Naive Algorithm:**



# A case study

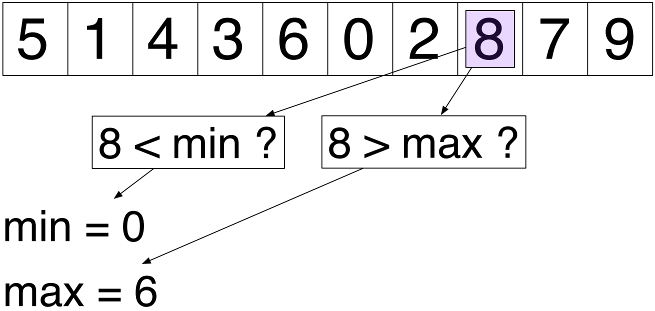
Find both the min. and the max. of an array of size $n$.

**Naive Algorithm:**



Find both the min. and the max. of an array of size $n$.

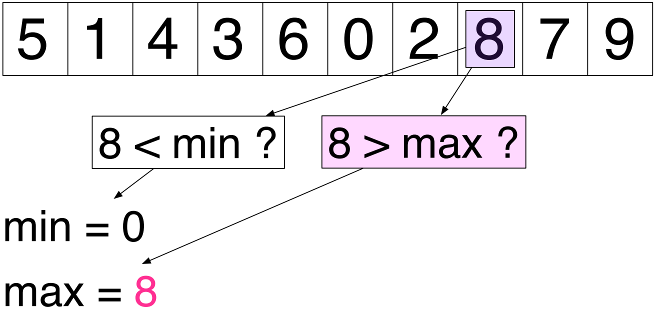
**Naive Algorithm:**



# A case study

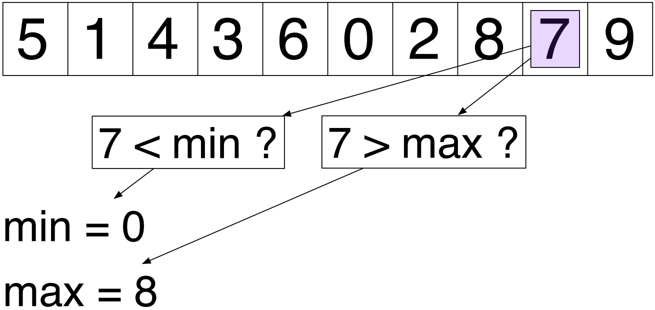
Find both the min. and the max. of an array of size $n$.

**Naive Algorithm:**



Find both the min. and the max. of an array of size $n$.

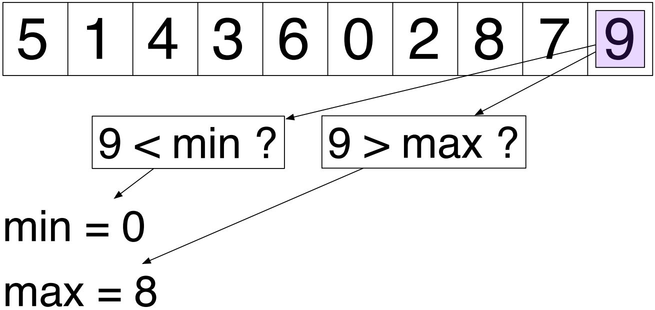
**Naive Algorithm:**



# A case study

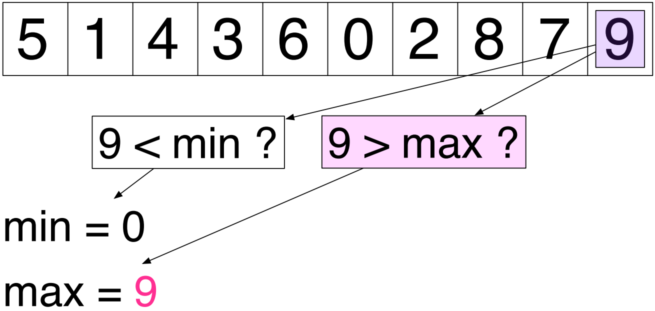
Find both the min. and the max. of an array of size $n$.

**Naive Algorithm:**



Find both the min. and the max. of an array of size $n$.

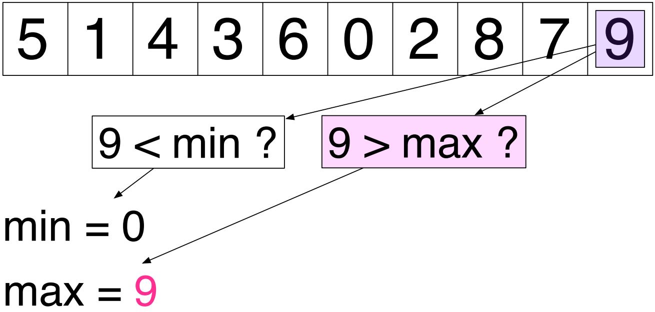
**Naive Algorithm:**



# A case study

Find both the min. and the max. of an array of size $n$.

**Naive Algorithm:** **$2n$ comparisons**




**Can we do better?**

# A case study

Find both the min. and the max. of an array of size $n$.

**Optimized Algorithm:**

| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| 5 | 1 | 4 | 3 | 6 | 0 | 2 | 8 | 7 | 9 |

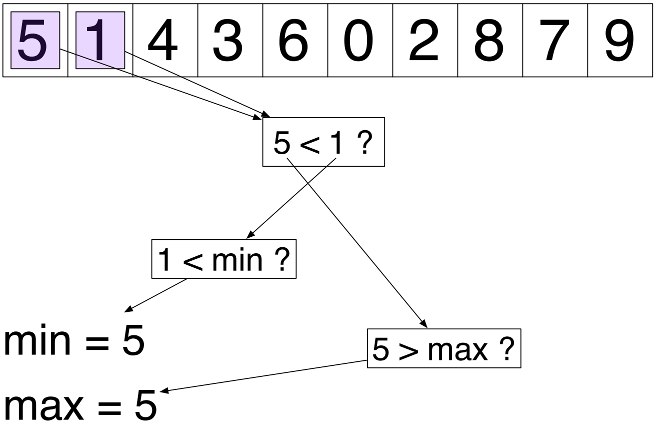
min = 5

max = 5

# A case study

Find both the min. and the max. of an array of size $n$.

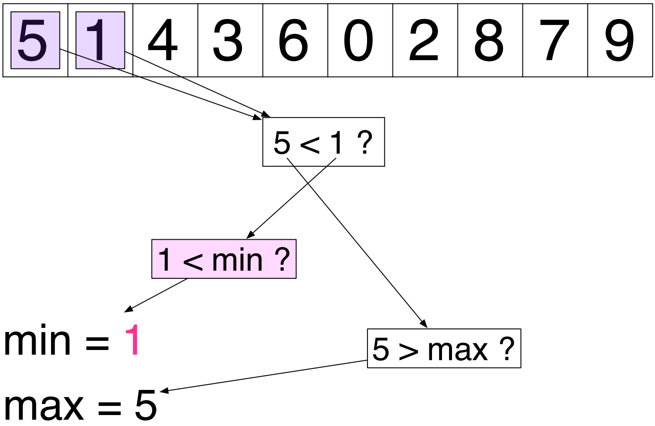
**Optimized Algorithm:**



# A case study

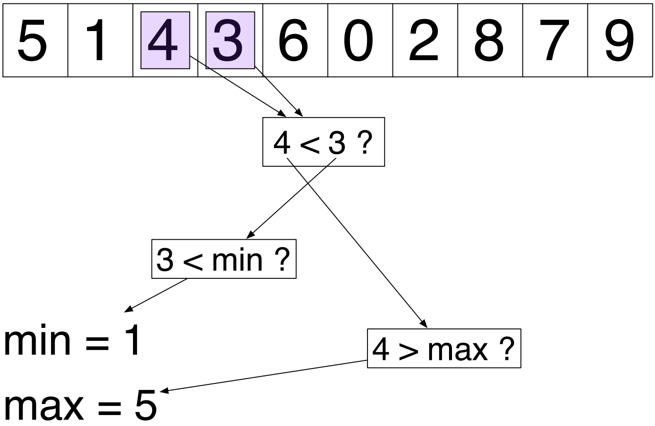
Find both the min. and the max. of an array of size $n$.

**Optimized Algorithm:**



Find both the min. and the max. of an array of size $n$.

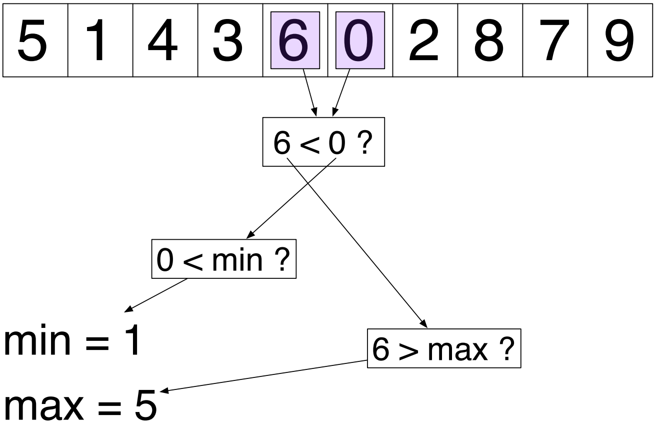
**Optimized Algorithm:**



# A case study

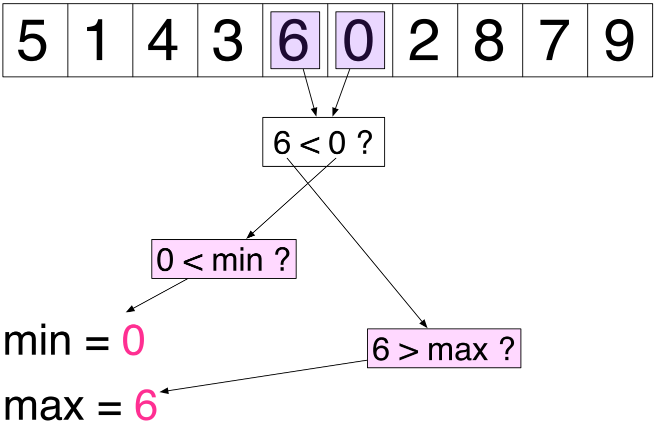
Find both the min. and the max. of an array of size $n$.

**Optimized Algorithm:**



Find both the min. and the max. of an array of size $n$.

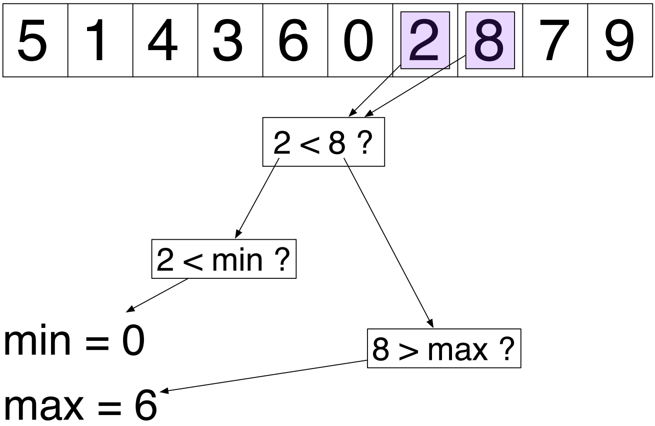
**Optimized Algorithm:**



# A case study

Find both the min. and the max. of an array of size $n$.

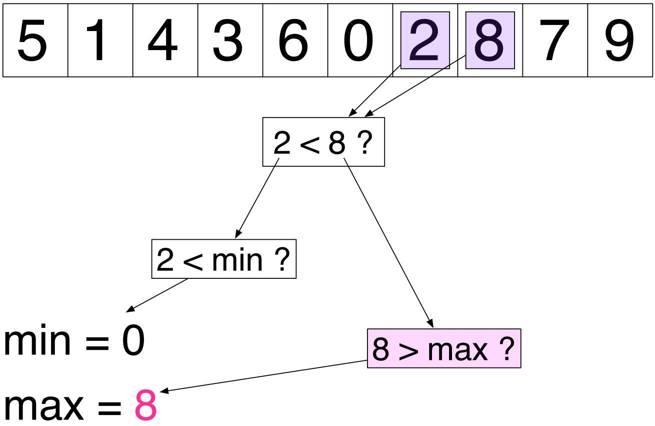
**Optimized Algorithm:**



# A case study

Find both the min. and the max. of an array of size $n$.

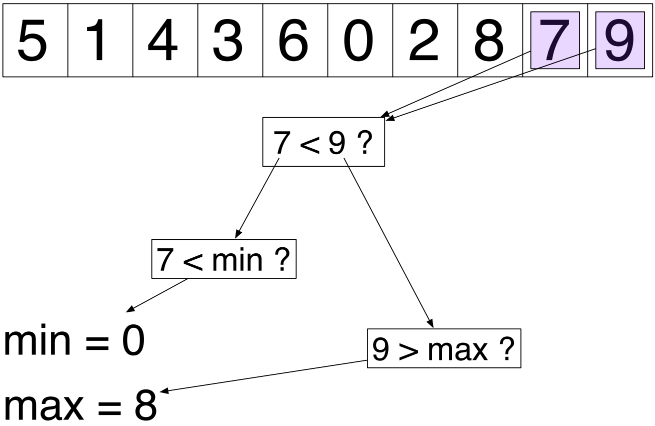
**Optimized Algorithm:**



# A case study

Find both the min. and the max. of an array of size $n$.

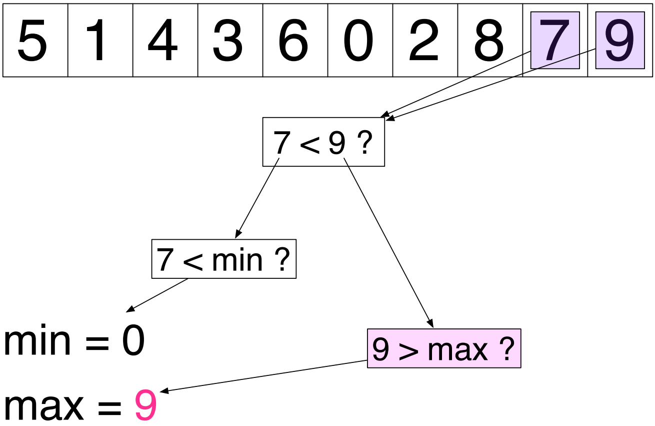
**Optimized Algorithm:**



# A case study

Find both the min. and the max. of an array of size $n$.

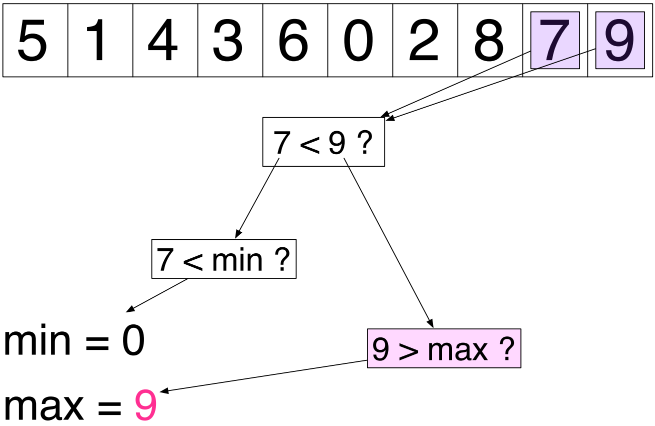
**Optimized Algorithm:**



# A case study

Find both the min. and the max. of an array of size $n$.

**Optimized Algorithm:** **$3n/2$ comparisons (optimal)**



# A case study

Find both the min. and the max. of an array of size $n$.

**Optimized Algorithm:** **$3n/2$ comparisons (optimal)**

**Naive Algorithm:** **$2n$ comparisons**

# A case study

Find both the min. and the max. of an array of size $n$.

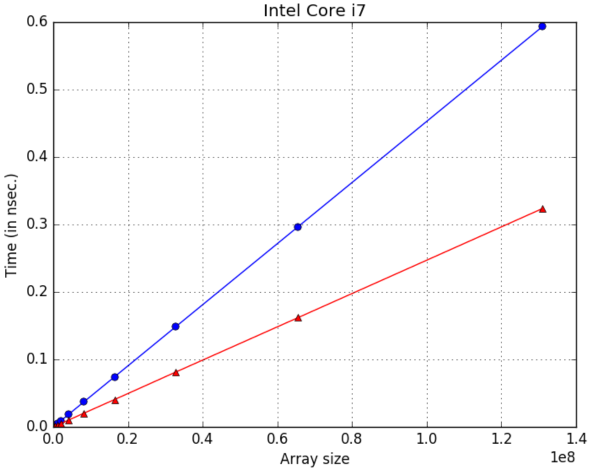
**Optimized Algorithm:** **$3n/2$ comparisons (optimal)**

**Naive Algorithm:** **$2n$ comparisons**

**In practice, on uniform random data?**

# A case study

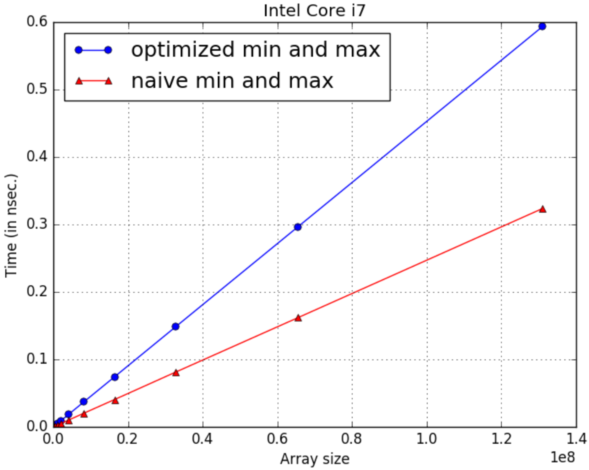
Find both the min. and the max. of an array of size $n$.




- in C,
- using `gcc -O0`,
- random integers

# A case study

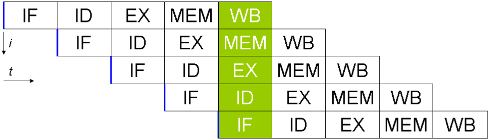
Find both the min. and the max. of an array of size $n$.




- in C,
- using `gcc -O0`,
- random integers

# Some background (mostly from Hennessy and Patterson [HP11])

- Most modern processors are pipelined
- Instructions are parallelized

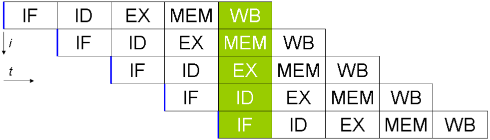



**Branch predictors are used to avoid stalls on branches!**

- Conditional instructions (such as the “if” statement) yield branches in the execution of a program
- The **branch predictor** will guess which branch will be *taken* (T) or not (NT).
- A **misprediction** can be quite **expensive**!
- Different schemes: static, dynamic, local, global,...

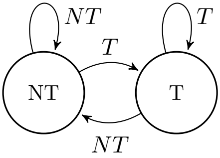
# Some background (mostly from Hennessy and Patterson [HP11])

- Most modern processors are pipelined
- Instructions are parallelized

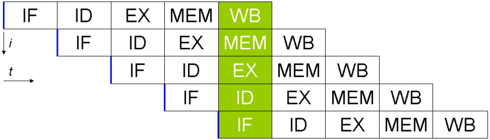



**Branch predictors are used to avoid stalls on branches!**

1-bit predictor:

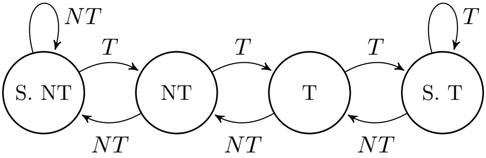


- Most modern processors are pipelined
- Instructions are parallelized

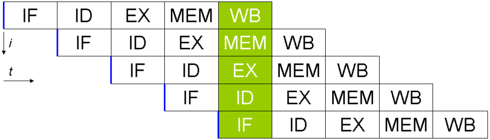



**Branch predictors are used to avoid stalls on branches!**

2-bit predictor:

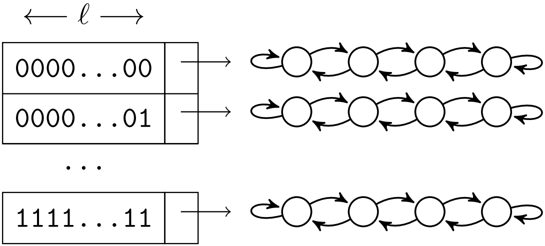


- Most modern processors are pipelined
- Instructions are parallelized




**Branch predictors are used to avoid stalls on branches!**

Global (or mixed) predictor:



- Most modern processors are pipelined
- Instructions are parallelized

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| IF | ID | EX | MEM | WB | | | | |
| | IF | ID | EX | MEM | WB | | | |
| | | IF | ID | EX | MEM | WB | | |
| | | | IF | ID | EX | MEM | WB | |
| | | | | IF | ID | EX | MEM | WB |

**Branch predictors are used to avoid stalls on branches!**

- Conditional instructions (such as the "if" statement) yield branches in the execution of a program
- The **branch predictor** will guess which branch will be *taken* (T) or not (NT).
- A **misprediction** can be quite **expensive**!
- Different schemes: static, dynamic, local, global,...
- Min and max search is very sensitive to branch prediction...

# Back to simultaneous min and max search

## Proposition

*Expected number of mispredictions, for the uniform distribution, on arrays of size n:*

- **Naive Min Max Search:**
  - $\sim 4 \log n$ *for the 1-bit predictor*
  - $\sim 2 \log n$ *for the two 2-bit predictors and the 3-bit saturating counter.*
- **Optimized Min Max Search:**
  - $\sim n/4 + \mathcal{O}(\log n)$ *for all four predictors.*

Idea of the proof:

- asymptotic analysis of the records in a random permutation,
- use the fundamental bijection that relates the records to the cycles in permutations,
- use classical results on the average number of cycles.

# We are not alone...

- Brodal & Moruz, 2005 : mispredictions and (adaptive) sorting

## Tradeoffs Between Branch Mispredictions and Comparisons for Sorting Algorithms

Gerth Stølting Brodal[1,*] and Gabriel Moruz[1]

BRICS**, Department of Computer Science, University of Aarhus,
IT Parken, Åbogade 34, DK-8200 Århus N, Denmark
{gerth, gabi}@daimi.au.dk

**Abstract.** Branch mispredictions is an important factor affecting the running time in practice. In this paper we consider tradeoffs between the number of branch mispredictions and the number of comparisons for

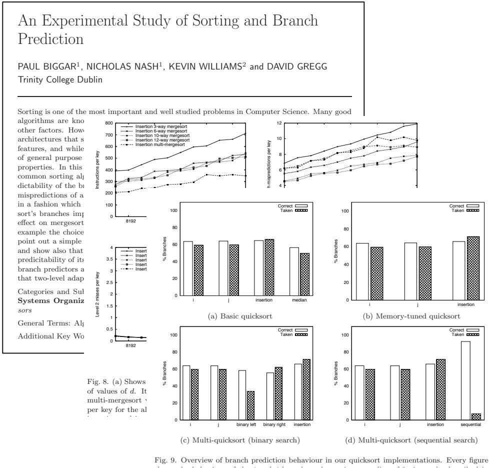
sorting
algorith
mispred
by adop
tions. F
rithm p
$\Omega(n \log$
of inver
by Esti
col and
mispred

| Measure | Comparisons | Branch mispredictions |
|---|---|---|
| Dis | $O(dn(1+\log(1+\mathrm{Dis})))$ | $\Omega(n\log_d(1+\mathrm{Dis}))$ |
| Exc | $O(dn(1+\mathrm{Exc}\log(1+\mathrm{Exc})))$ | $\Omega(n\mathrm{Exc}\log_d(1+\mathrm{Exc}))$ |
| Enc | $O(dn(1+\log(1+\mathrm{Enc})))$ | $\Omega(n\log_d(1+\mathrm{Enc}))$ |
| Inv | $O(dn(1+\log(1+\mathrm{Inv}/n)))$ | $\Omega(n\log_d(1+\mathrm{Inv}/n))$ |
| Max | $O(dn(1+\log(1+\mathrm{Max})))$ | $\Omega(n\log_d(1+\mathrm{Max}))$ |
| Osc | $O(dn(1+\log(1+\mathrm{Osc}/n)))$ | $\Omega(n\log_d(1+\mathrm{Osc}/n))$ |
| Reg | $O(dn(1+\log(1+\mathrm{Reg})))$ | $\Omega(n\log_d(1+\mathrm{Reg}))$ |
| Rem | $O(dn(1+\mathrm{Rem}\log(1+\mathrm{Rem})))$ | $\Omega(n\mathrm{Rem}\log_d(1+\mathrm{Rem}))$ |
| Runs | $O(dn(1+\log(1+\mathrm{Runs})))$ | $\Omega(n\log_d(1+\mathrm{Runs}))$ |
| SMS | $O(dn(1+\log(1+\mathrm{SMS})))$ | $\Omega(n\log_d(1+\mathrm{SMS}))$ |
| SUS | $O(dn(1+\log(1+\mathrm{SUS})))$ | $\Omega(n\log_d(1+\mathrm{SUS}))$ |

**Fig. 4.** Lower bounds on the number of branch mispredictions for deterministic comparison based adaptive sorting algorithms for different measures of presortedness, given the upper bounds on the number of comparisons

# We are not alone...

- Brodal & Moruz, 2005 : mispredictions and (adaptive) sorting
- Biggar *et al*, 2008 : experimental, branch prediction and sorting


An Experimental Study of Sorting and Branch Prediction

PAUL BIGGAR[1], NICHOLAS NASH[1], KEVIN WILLIAMS[2] and DAVID GREGG
Trinity College Dublin

Sorting is one of the most important and well studied problems in Computer Science. Many good
algorithms are kno
other factors. How
architectures that s
features, and while
of general purpose
properties. In this
common sorting al
dictability of the b
mispredictions of a
in a fashion which
sort's branches im
effect on mergesort
example the choice
point out a simple
and show also that
predicitability of its
branch predictors a
that two-level adap

Categories and Sul
**Systems Organiz**
*sors*

General Terms: Al

Additional Key Wo

Fig. 8. (a) Shows
of values of *d*. It
multi-mergesort
per key for the al



(a) Basic quicksort

(b) Memory-tuned quicksort

(c) Multi-quicksort (binary search)

(d) Multi-quicksort (sequential search)

Fig. 9. Overview of branch prediction behaviour in our quicksort implementations. Every figure

# We are not alone...

- Brodal & Moruz, 2005 : mispredictions and (adaptive) sorting
- Biggar *et al*, 2008 : experimental, branch prediction and sorting
- Sander and Winkel, 2004 : quicksort variant without branches

## Super Scalar Sample Sort

Peter Sanders[1] and Sebastian Winkel[2]

[1] Max Planck Institut für Informatik
Saarbrücken, Germany, sanders@mpi-sb.mpg.de
[2] Chair for Prog. Lang. and Compiler Construction
Saarland University, Saarbrücken, Germany, sewi@cs.uni-sb.de

**Abstract.** Sample sort, a generalization of quicksort that partitions the input into many pieces, is known as the best practical comparison based sorting algorithm for distributed memory parallel computers. We show that … mic i … cond … facili … final … ber c … Itani … the ( … quick …

$t := \langle s_{k/2}, s_{k/4}, s_{3k/4}, s_{k/8}, s_{3k/8}, s_{5k/8}, s_{7k/8}, \ldots \rangle$ //
**for** $i := 1$ **to** $n$ **do** // locate each element
  $j := 1$ // current tree node := root
  **repeat** $\log k$ *times* // will be unrolled
    $j := 2j + (a_i > t_j)$ // left or right?
  $j := j - k + 1$ // bucket index
  $|b_j|{+}{+}$ // count bucket size
  $o(i) := j$ // remember oracle

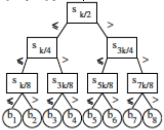



**Fig. 2.** Finding buckets using implicit search trees. The picture is for $k = 8$. We adopt the C convention that "$x > y$" is one if $x > y$ holds, and zero else.

# We are not alone...

- Brodal & Moruz, 2005 : mispredictions and (adaptive) sorting
- Biggar *et al*, 2008 : experimental, branch prediction and sorting
- Sander and Winkel, 2004 : quicksort variant without branches
- Elmasry *et al*, 2012 : mergesort variant without branches

**Branch Mispredictions Don't Affect Mergesort***


Amr Elmasry[1], Jyrki Katajainen[1,2], and Max Stenmark[2]

[1] Department of Computer Science, University of Copenhagen
Universitetsparken 1, 2100 Copenhagen East, Denmark
[2] Jyrki Katajainen and Company
Thorsgade 101, 2200 Copenhagen North, Denmark



**Abstract.** In quicksort, due to branch mispredictions, a skewed pivot-selection strategy can lead to a better performance than the exact-median pivot-selection strategy, e
free. In this paper we investigate t
the behaviour of mergesort. By d
branches, we can avoid most nega
dictions. When sorting a sequence
mergesort performs $n \log_2 n + O(n$
most $O(n)$ branch mispredictions


```
while (p != t1 && q != t2) {
  if (less(*q, *p)) {
    s = q;
    ++q;
  }
  else {
```

```
test:
  done = (q == t2);
  if (done) goto exit;
entrance:
  x = *p;
  s = p + 1;
  y = *q;
  t = q + 1;
  smaller = less(y, x);
  if (smaller) s = t;
  if (smaller) q = t;
  if (! smaller) p = s;
  if (! smaller) y = x;
  x = *r;
  *r = y;
  --s;
  *s = x;
  ++r;
  done = (p == t1);
  if (! done) goto test;
exit:
```

**Table 3.** The execution time [ns], the number of conditional branches, and the number of mispredictions, each per $n \log_2 n$, for two in-situ variants of mergesort.

| Program | In-situ `std::stable_sort` | | | | In-situ mergesort | | | |
|---|---|---|---|---|---|---|---|---|
| | Time | | Branches | Mispredicts | Time | | Branches | Mispredicts |
| $n$ | Per | Ares | | | Per | Ares | | |
| $2^{10}$ | 49.2 | 29.7 | 9.0 | 2.08 | 7.3 | 5.7 | 1.93 | 0.26 |
| $2^{15}$ | 57.6 | 35.0 | 11.1 | 2.38 | 7.1 | 5.6 | 1.94 | 0.15 |
| $2^{20}$ | 62.7 | 38.5 | 12.9 | 2.53 | 7.4 | 5.7 | 1.92 | 0.11 |
| $2^{25}$ | 68.0 | 41.3 | 14.4 | 2.62 | 7.6 | 5.7 | 1.92 | 0.09 |

# We are not alone...

- Brodal & Moruz, 2005 : mispredictions and (adaptive) sorting
- Biggar *et al*, 2008 : experimental, branch prediction and sorting
- Sander and Winkel, 2004 : quicksort variant without branches
- Elmasry *et al*, 2012 : mergesort variant without branches
- Kaligosi and Peter Sanders, 2006 : mispredictions and quicksort

**How Branch Mispredictions Affect Quicksort**

Kanela Kaligosi[1] and Peter Sanders[2]

[1] Max Planck In
Saarbrüc
kaligosi@
[2] Universität K
sanders

**Abstract.** We explain the cou
“good” pivots (close to the medi
not impro
pivot *imp*
count dec
direction
fect of sin
hardware.

**Table 1.** Number of branch mispredictions

| | random pivot | $\alpha$-skewed pivot |
|---|---|---|
| static predictor | $\frac{\ln 2}{2} n \lg n + \mathcal{O}(n)$, $\frac{\ln 2}{2} \approx 0.3466$ | $\frac{\alpha}{H(\alpha)} n \lg n + \mathcal{O}(n)$, $\alpha < 1/2$<br>$\frac{1-\alpha}{H(\alpha)} n \lg n + \mathcal{O}(n)$, $\alpha \geq 1/2$ |
| 1-bit predictor | $\frac{2 \ln 2}{3} n \lg n + \mathcal{O}(n)$, $\frac{2 \ln 2}{3} \approx 0.4621$ | $\frac{2\alpha(1-\alpha)}{H(\alpha)} n \lg n + \mathcal{O}(n)$ |
| 2-bit predictor | $\frac{28 \ln 2}{45} n \lg n + \mathcal{O}(n)$, $\frac{28 \ln 2}{45} \approx 0.4313$ | $\frac{2\alpha^4 - 4\alpha^3 + \alpha^2 + \alpha}{(1-\alpha(1-\alpha))H(\alpha)} n \lg n + \mathcal{O}(n)$ |

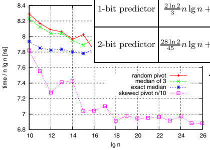
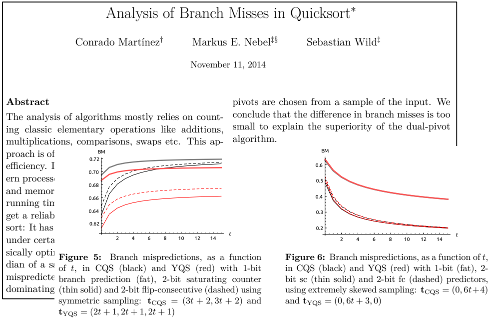



**Fig. 3.** Time / $n \lg n$ for random pivot, median of 3, exact median, 1/10-skewed pivot

# We are not alone...

- Brodal & Moruz, 2005 : mispredictions and (adaptive) sorting
- Biggar *et al*, 2008 : experimental, branch prediction and sorting
- Sander and Winkel, 2004 : quicksort variant without branches
- Elmasry *et al*, 2012 : mergesort variant without branches
- Kaligosi and Peter Sanders, 2006 : mispredictions and quicksort
- Martínez, Nebel and Wild, 2014 : mispredictions and quicksort



Analysis of Branch Misses in Quicksort*

Conrado Martínez[†] Markus E. Nebel[‡§] Sebastian Wild[‡]

November 11, 2014

**Abstract**

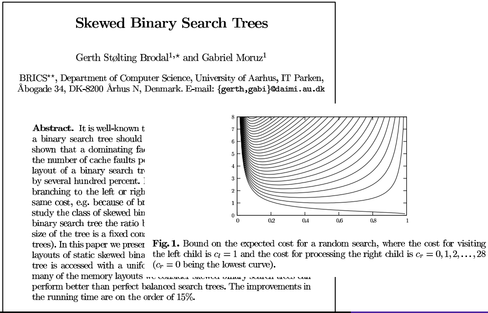
The analysis of algorithms mostly relies on counting classic elementary operations like additions, multiplications, comparisons, swaps etc. This approach is of efficiency. I ern process and memor running tin get a reliab sort: It has under certa sically opti dian of a s mispredicte dominating pivots are chosen from a sample of the input. We conclude that the difference in branch misses is too small to explain the superiority of the dual-pivot algorithm.

**Figure 5:** Branch mispredictions, as a function of $t$, in CQS (black) and YQS (red) with 1-bit branch prediction (fat), 2-bit saturating counter (thin solid) and 2-bit flip-consecutive (dashed) using symmetric sampling: $\mathbf{t}_{\text{CQS}} = (3t+2, 3t+2)$ and $\mathbf{t}_{\text{YQS}} = (2t+1, 2t+1, 2t+1)$

**Figure 6:** Branch mispredictions, as a function of $t$, in CQS (black) and YQS (red) with 1-bit (fat), 2-bit sc (thin solid) and 2-bit fc (dashed) predictors, using extremely skewed sampling: $\mathbf{t}_{\text{CQS}} = (0, 6t+4)$ and $\mathbf{t}_{\text{YQS}} = (0, 6t+3, 0)$

# We are not alone...

- Brodal & Moruz, 2005 : mispredictions and (adaptive) sorting
- Biggar *et al*, 2008 : experimental, branch prediction and sorting
- Sander and Winkel, 2004 : quicksort variant without branches
- Elmasry *et al*, 2012 : mergesort variant without branches
- Kaligosi and Peter Sanders, 2006 : mispredictions and quicksort
- Martínez, Nebel and Wild, 2014 : mispredictions and quicksort
- Brodal and Moruz, 2006 : skewed binary search trees



**Skewed Binary Search Trees**

Gerth Stølting Brodal[1,*] and Gabriel Moruz[1]

BRICS**, Department of Computer Science, University of Aarhus, IT Parken, Åbogade 34, DK-8200 Århus N, Denmark. E-mail: {gerth,gabi}@daimi.au.dk

**Abstract.** It is well-known t a binary search tree should shown that a dominating fa the number of cache faults p layout of a binary search tr by several hundred percent. branching to the left or righ same cost, e.g. because of br study the class of skewed bin binary search tree the ratio b size of the tree is a fixed con trees). In this paper we preser layouts of static skewed bina tree is accessed with a unifo many of the memory layouts perform better than perfect balanced search trees. The improvements in the running time are on the order of 15%.

**Fig. 1.** Bound on the expected cost for a random search, where the cost for visiting the left child is $c_l = 1$ and the cost for processing the right child is $c_r = 0, 1, 2, \ldots, 28$ ($c_r = 0$ being the lowest curve).

# Introducing unnecessary tests to speed up

POW(x,n)

```
r = 1;
while (n > 0) {
  // n is odd
  if (n & 1)
    r = r * x;
  n /= 2;
  x = x * x;
}
```

`x` is a floating-point number, `n` is an integer and `r` is the result.

$x^n = (x^2)^{\lfloor n/2 \rfloor} x^{n_0}$

# Introducing unnecessary tests to speed up

POW(x,n)

```
r = 1;
while (n > 0) {
  // n is odd
  if (n & 1) ℙ = 1/2
    r = r * x;
  n /= 2;
  x = x * x;
}
```

x is a floating-point number, n is an integer and r is the result.

$x^n = (x^2)^{\lfloor n/2 \rfloor} x^{n_0}$

# Introducing unnecessary tests to speed up

POW(x,n)

```
r = 1;
while (n > 0) {
  // n is odd
  if (n & 1)  ℙ = 1/2
    r = r * x;
  n /= 2;
  x = x * x;
}
```

`x` is a floating-point number, `n` is an integer and `r` is the result.

$x^n = (x^2)^{\lfloor n/2 \rfloor} x^{n_0}$

UNROLLED(x,n)

```
r = 1;
while (n > 0) {
  t = x * x;
  // n0 == 1
  if (n & 1)
    r = r * x;
  // n1 == 1
  if (n & 2)
    r = r * t;
  n /= 4;
  x = t * t;
}
```

$x^n = (x^4)^{\lfloor n/4 \rfloor} (x^2)^{n_1} x^{n_0}$

# Introducing unnecessary tests to speed up

POW(x,n)

```
r = 1;
while (n > 0) {
  // n is odd
  if (n & 1)   ℙ = 1/2
    r = r * x;
  n /= 2;
  x = x * x;
}
```

x is a floating-point number, n is an integer and r is the result.

$x^n = (x^2)^{\lfloor n/2 \rfloor} x^{n_0}$

UNROLLED(x,n)

```
r = 1;
while (n > 0) {
  t = x * x;
  // n0 == 1
  if (n & 1)   ℙ = 1/2
    r = r * x;
  // n1 == 1
  if (n & 2)   ℙ = 1/2
    r = r * t;
  n /= 4;
  x = t * t;
}
```

$x^n = (x^4)^{\lfloor n/4 \rfloor} (x^2)^{n_1} x^{n_0}$

# Introducing unnecessary tests to speed up

POW(x,n)

```
r = 1;
while (n > 0) {
  // n is odd
  if (n & 1)    ℙ = 1/2
    r = r * x;
  n /= 2;
  x = x * x;
}
```

x is a floating-point number, n is an integer and r is the result.

$x^n = (x^2)^{\lfloor n/2 \rfloor} x^{n_0}$

UNROLLED(x,n)

```
r = 1;
while (n > 0) {
  t = x * x;
  // n0 == 1
  if (n & 1)    ℙ = 1/2
    r = r * x;
  // n1 == 1
  if (n & 2)    ℙ = 1/2
    r = r * t;
  n /= 4;
  x = t * t;
}
```

$x^n = (x^4)^{\lfloor n/4 \rfloor} (x^2)^{n_1} x^{n_0}$

GUIDED(x,n)

```
r = 1;
while (n > 0) {
  t = x * x;
  // n1n0! = 00
  if (n & 3){
    if (n & 1)
      r = r * x;
    if (n & 2)
      r = r * t;
  }
  n /= 4;
  x = t * t;
}
```

# Introducing unnecessary tests to speed up

POW(x,n)

```
r = 1;
while (n > 0) {
  // n is odd
  if (n & 1)        ℙ = 1/2
    r = r * x;
  n /= 2;
  x = x * x;
}
```

x is a floating-point number, n is an integer and r is the result.

$$x^n = (x^2)^{\lfloor n/2 \rfloor} x^{n_0}$$

UNROLLED(x,n)

```
r = 1;
while (n > 0) {
  t = x * x;
  // n0 == 1
  if (n & 1)        ℙ = 1/2
    r = r * x;
  // n1 == 1
  if (n & 2)        ℙ = 1/2
    r = r * t;
  n /= 4;
  x = t * t;
}
```

$$x^n = (x^4)^{\lfloor n/4 \rfloor} (x^2)^{n_1} x^{n_0}$$

GUIDED(x,n)

```
r = 1;
while (n > 0) {
  t = x * x;
  // n1n0! = 00
  if (n & 3){       ℙ = 3/4
    if (n & 1)
      r = r * x;
    if (n & 2)
      r = r * t;
  }
  n /= 4;
  x = t * t;
}
```

# Introducing unnecessary tests to speed up

POW(x,n)

```
r = 1;
while (n > 0) {
  // n is odd
  if (n & 1)    ℙ = 1/2
    r = r * x;
  n /= 2;
  x = x * x;
}
```

x is a floating-point number, n is an integer and r is the result.

$x^n = (x^2)^{\lfloor n/2 \rfloor} x^{n_0}$

UNROLLED(x,n)

```
r = 1;
while (n > 0) {
  t = x * x;
  // n_0 == 1
  if (n & 1)    ℙ = 1/2
    r = r * x;
  // n_1 == 1
  if (n & 2)    ℙ = 1/2
    r = r * t;
  n /= 4;
  x = t * t;
}
```

$x^n = (x^4)^{\lfloor n/4 \rfloor} (x^2)^{n_1} x^{n_0}$

GUIDED(x,n)

```
r = 1;
while (n > 0) {
  t = x * x;
  // n_1 n_0 != 00
  if (n & 3){    ℙ = 3/4
    if (n & 1)    ℙ = 2/3
      r = r * x;
    if (n & 2)    ℙ = 2/3
      r = r * t;
  }
  n /= 4;
  x = t * t;
}
```

# Introducing unnecessary tests to speed up

POW(x,n)

```
r = 1;
while (n > 0) {
  // n is odd
  if (n & 1)  ℙ = 1/2
    r = r * x;
  n /= 2;
  x = x * x;
}
```

x is a floating-point number, n is an integer and r is the result.

$x^n = (x^2)^{\lfloor n/2 \rfloor} x^{n_0}$

UNROLLED(x,n)

```
r = 1;
while (n > 0) {
  t = x * x;
  // n0 == 1
  if (n & 1)  ℙ = 1/2
    r = r * x;
  // n1 == 1
  if (n & 2)  ℙ = 1/2
    r = r * t;
  n /= 4;
  x = t * t;
}
```

$x^n = (x^4)^{\lfloor n/4 \rfloor} (x^2)^{n_1} x^{n_0}$

GUIDED(x,n)

```
r = 1;
while (n > 0) {
  t = x * x;
  // n1n0 != 00
  if (n & 3){  ℙ = 3/4
    if (n & 1)  ℙ = 2/3
      r = r * x;
    if (n & 2)  ℙ = 2/3
      r = r * t;
  }
  n /= 4;
  x = t * t;
}
```

- 25 % more comparisons for GUIDED than for UNROLLED

# Introducing unnecessary tests to speed up

POW(x,n)

```
r = 1;
while (n > 0) {
  // n is odd
  if (n & 1)    ℙ = 1/2
    r = r * x;
  n /= 2;
  x = x * x;
}
```

x is a floating-point number, n is an integer and r is the result.

$x^n = (x^2)^{\lfloor n/2 \rfloor} x^{n_0}$

UNROLLED(x,n)

```
r = 1;
while (n > 0) {
  t = x * x;
  // n0 == 1
  if (n & 1)    ℙ = 1/2
    r = r * x;
  // n1 == 1
  if (n & 2)    ℙ = 1/2
    r = r * t;
  n /= 4;
  x = t * t;
}
```

$x^n = (x^4)^{\lfloor n/4 \rfloor} (x^2)^{n_1} x^{n_0}$

GUIDED(x,n)

```
r = 1;
while (n > 0) {
  t = x * x;
  // n1n0! = 00
  if (n & 3){    ℙ = 3/4
    if (n & 1)    ℙ = 2/3
      r = r * x;
    if (n & 2)    ℙ = 2/3
      r = r * t;
  }
  n /= 4;
  x = t * t;
}
```

- 25 % more comparisons for GUIDED than for UNROLLED
- GUIDED exponential is 14% faster than the UNROLLED one;

# Introducing unnecessary tests to speed up

POW(x,n)

```
r = 1;
while (n > 0) {
  // n is odd
  if (n & 1)   ℙ = 1/2
    r = r * x;
  n /= 2;
  x = x * x;
}
```

x is a floating-point number, n is an integer and r is the result.

$x^n = (x^2)^{\lfloor n/2 \rfloor} x^{n_0}$

UNROLLED(x,n)

```
r = 1;
while (n > 0) {
  t = x * x;
  // n_0 == 1
  if (n & 1)   ℙ = 1/2
    r = r * x;
  // n_1 == 1
  if (n & 2)   ℙ = 1/2
    r = r * t;
  n /= 4;
  x = t * t;
}
```

$x^n = (x^4)^{\lfloor n/4 \rfloor} (x^2)^{n_1} x^{n_0}$

GUIDED(x,n)

```
r = 1;
while (n > 0) {
  t = x * x;
  // n_1 n_0 != 00
  if (n & 3){   ℙ = 3/4
    if (n & 1)   ℙ = 2/3
      r = r * x;
    if (n & 2)   ℙ = 2/3
      r = r * t;
  }
  n /= 4;
  x = t * t;
}
```

- 25 % more comparisons for GUIDED than for UNROLLED
- GUIDED exponential is 14% faster than the UNROLLED one;
- GUIDED exponential is 29% faster than the classical one;

# Introducing unnecessary tests to speed up

POW(x,n)

```
r = 1;
while (n > 0) {
  // n is odd
  if (n & 1) ℙ = 1/2
    r = r * x;
  n /= 2;
  x = x * x;
}
```

x is a floating-point number, n is an integer and r is the result.

$x^n = (x^2)^{\lfloor n/2 \rfloor} x^{n_0}$

UNROLLED(x,n)

```
r = 1;
while (n > 0) {
  t = x * x;
  // n0 == 1
  if (n & 1) ℙ = 1/2
    r = r * x;
  // n1 == 1
  if (n & 2) ℙ = 1/2
    r = r * t;
  n /= 4;
  x = t * t;
}
```

$x^n = (x^4)^{\lfloor n/4 \rfloor} (x^2)^{n_1} x^{n_0}$

GUIDED(x,n)

```
r = 1;
while (n > 0) {
  t = x * x;
  // n1n0! = 00
  if (n & 3){ ℙ = 3/4
    if (n & 1) ℙ = 2/3
      r = r * x;
    if (n & 2) ℙ = 2/3
      r = r * t;
  }
  n /= 4;
  x = t * t;
}
```

- 25 % more comparisons for GUIDED than for UNROLLED
- GUIDED exponential is 14% faster than the UNROLLED one;
- GUIDED exponential is 29% faster than the classical one;
- yet, the number of multiplications is essentially the same.

# Guided Pow: average number of mispredictions

## Theorem

*Compute $x^n$, for random $n$ in $\{0, \ldots, N-1\}$.*

- **Expected nb. of conditionals:**

  $\sim \log_2 N$ *for classical and unrolled pow*

  $\sim \frac{5}{4} \log_2 N$ *for the guided one*

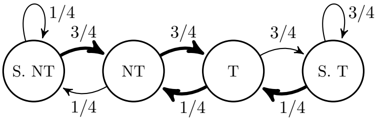
- **Expected nb. of mispredictions:**

  $\sim \frac{1}{2} \log_2 N$ *for classical and unrolled pow*

  $\sim \left(\frac{1}{2}\mu(\frac{3}{4}) + \frac{3}{4}\mu(\frac{2}{3})\right) \log_2 N$ *for guided pow*

GUIDED(x,n)

```
r = 1;
while (n > 0) {
  t = x * x;
  // n1n0! = 00
  if (n & 3) {
    if (n & 1)
      r = r * x;
    if (n & 2)
      r = r * t;
  }
  n /= 4;
  x = t * t;
}
```




$$\mu(\tfrac{3}{4}) = \tfrac{3}{10} \text{ and } \mu(\tfrac{2}{3}) = \tfrac{2}{5}$$

- 25 % more comparisons than unrolled
- unnecessary if : added mispred.
- other ones : less mispred.

▶ 5 % less mispred. (2-bit predictor)

▶ **11 % less mispred.** (3-bit predictor)

# Guided Pow: average number of mispredictions

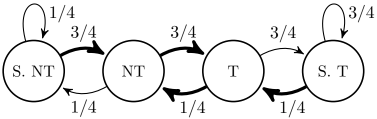
## Theorem

*Compute $x^n$, for random $n$ in $\{0, \dots, N-1\}$.*

- **Expected nb. of conditionals:**
  - $\sim \log_2 N$ *for classical and unrolled pow*
  - $\sim \frac{5}{4} \log_2 N$ *for the guided one*
- **Expected nb. of mispredictions:**
  - $\sim \frac{1}{2} \log_2 N$ *for classical and unrolled pow*
  - $\sim 0.45 \log_2 N$ *for guided pow (2-bit pred.)*

GUIDED(x,n)

```
r = 1;
while (n > 0) {
  t = x * x;
  // n1n0 != 00
  if (n & 3) {
    if (n & 1)
      r = r * x;
    if (n & 2)
      r = r * t;
  }
  n /= 4;
  x = t * t;
}
```




$\mu(\frac{3}{4}) = \frac{3}{10}$ and $\mu(\frac{2}{3}) = \frac{2}{5}$

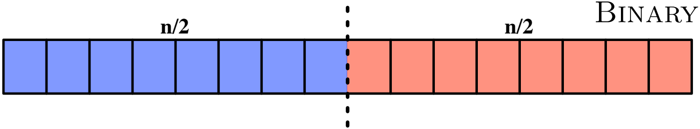
- 25 % more comparisons than unrolled
- unnecessary if : added mispred.
- other ones : less mispred.

▶ 5 % less mispred. (2-bit predictor)

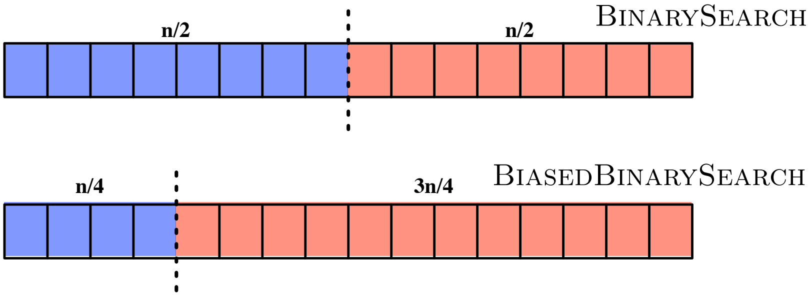
▶ **11 % less mispred.** (3-bit predictor)

# Unbalancing the binary search

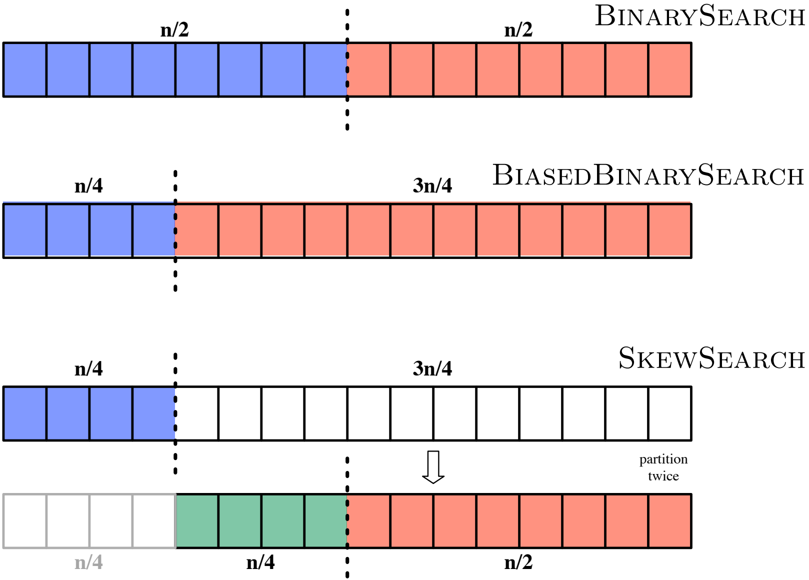
BinarySearch



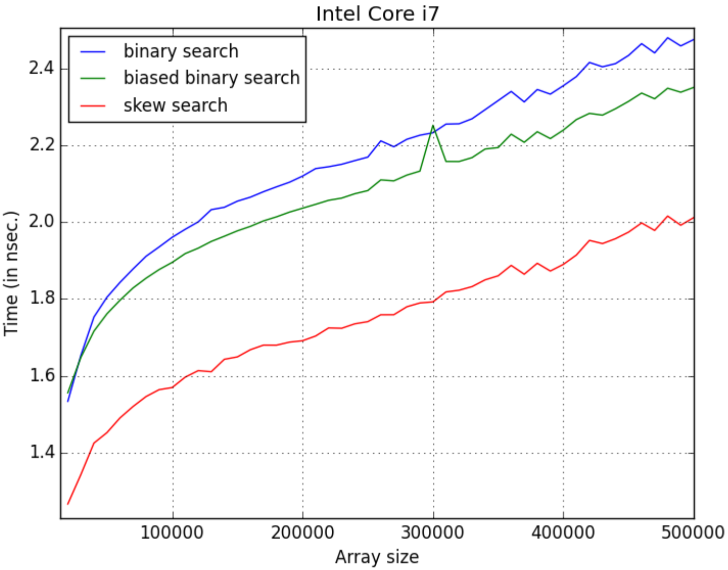
# Unbalancing the binary search



# Unbalancing the binary search



# Unbalancing the binary search



# Analysis of the local predictor

## Theorem

*For arrays of size $n$ filled with random uniform integers. $C_n$ is the number of comparisons and $M_n$ the number of mispredictions.*

| | BinarySearch | BiasedBinarySearch | SkewSearch |
|---|---|---|---|
| $\mathbb{E}[C_n]$ | $\frac{\log n}{\log 2}$ | $\frac{4 \log n}{(4 \log 4 - 3 \log 3)}$ | $\frac{7 \log n}{(6 \log 2)}$ |
| $\mathbb{E}[M_n]$ | $\frac{\log n}{(2 \log 2)}$ | $\mu(\frac{1}{4})\mathbb{E}[C_n]$ | $\left(\frac{4}{7}\mu(\frac{1}{4}) + \frac{3}{7}\mu(\frac{1}{3})\right)\mathbb{E}[C_n]$ |

*$\mu$ is the expected misprediction probability associated with the predictor.*

Idea of the proof:

- Get the expected number of times a given conditional is executed by Roura's Master Theorem [Rou01].
- Ensure that our predictors behave *almost* like Markov chains.

# Analysis of the local predictor

**Theorem**

*For arrays of size $n$ filled with random uniform integers. $C_n$ is the number of comparisons and $M_n$ the number of mispredictions.*

| | BinarySearch | BiasedBinarySearch | SkewSearch |
|---|---|---|---|
| $\mathbb{E}[C_n]$ | $1.44 \log n$ | $1.78 \log n$ | $1.68 \log n$ |
| $\mathbb{E}[M_n]$ | $0.72 \log n$ | $0.53 \log n$ | $0.58 \log n$ |

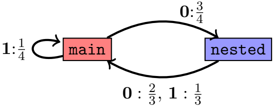
**with a 2-bit saturated counter.**

Idea of the proof:

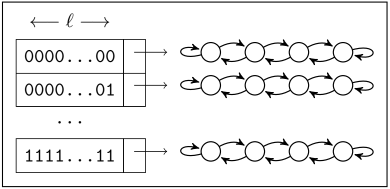
- Get the expected number of times a given conditional is executed by Roura's Master Theorem [Rou01].
- Ensure that our predictors behave *almost* like Markov chains.

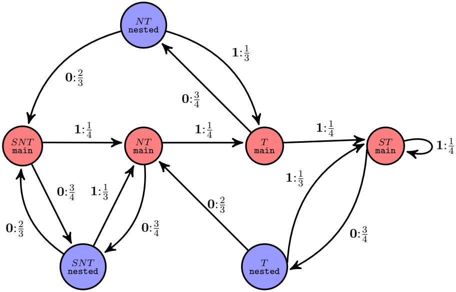
# What about a global predictor?

```
d = 0; f = n;
while (d < f){
    m1 = (3*d+f)/4;
    if (T[m1] > x) f = m1;
    else {
        m2 = (d+f)/2;
        if (T[m2] > x){
            f = m2;
            d = m1+1;
        }
        else d = m2+1;
    }
}
return f;
```




Global predictor



# Concluding remarks


Gerth Stølting Brodal and Gabriel Moruz.
Tradeoffs Between Branch Mispredictions and Comparisons for Sorting Algorithms.
In *Algorithms and Data Structures*, volume 3608, pages 385–395. Springer Berlin Heidelberg, 2005.

Gerth Stølting Brodal and Gabriel Moruz.
Skewed Binary Search Trees.
In *Algorithms ESA 2006*, volume 4168, pages 708–719. Springer Berlin Heidelberg, 2006.

Paul Biggar, Nicholas Nash, Kevin Williams, and David Gregg.
An experimental study of sorting and branch prediction.
*Journal of Experimental Algorithmics*, 12:1, June 2008.

John L. Hennessy and David A. Patterson.
*Computer Architecture, Fifth Edition: A Quantitative Approach*.
Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 5th edition, 2011.

Kanela Kaligosi and Peter Sanders.
How Branch Mispredictions Affect Quicksort.
In *Algorithms ESA 2006*, volume 4168, pages 780–791. Springer Berlin Heidelberg, 2006.

Conrado Martínez, Markus E. Nebel, and Sebastian Wild.
Analysis of branch misses in quicksort.
In *Proceedings of the Twelfth Workshop on Analytic Algorithmics and Combinatorics, ANALCO 2015, San Diego, CA, USA, January 4, 2015*, pages 114–128, 2015.

Salvador Roura.
Improved master theorems for divide-and-conquer recurrences.
*Journal of the ACM*, 48(2):170–205, 2001.