Realistic analysis of algorithms

Cyril Nicaud

LIGM – Université Paris-Est Marne-la-Vallée & CNRS

AofA 2019, june 25

Realistic?

1. Which algorithms are implemented in standard libraries and why?

2. How accurate is our model of computation? Can we improve it?

Realistic?

1. Which algorithms are implemented in standard libraries and why?
   - Java's Dual-Pivot Quicksort
     [C. Martínez, M. Nebel, R. Neininger, S. Wild, ...]
   - **TimSort (Python, Java, ...)**
   - ...
2. How accurate is our model of computation? Can we improve it?

Realistic?

1. Which algorithms are implemented in standard libraries and why?
   - Java's Dual-Pivot Quicksort
     [C. Martínez, M. Nebel, R. Neininger, S. Wild, ...]
   - **TimSort (Python, Java, ...)**
   - ...
2. How accurate is our model of computation? Can we improve it?
   - External memory model
   - Cache-oblivious model
   - **Branch predictions**
   - ...

I. TimSort

with N. Auger, V. Jugé & C. Pivoteau

TimSort algorithm

a	c	t	r	b	w	k	i	e	d	u	n

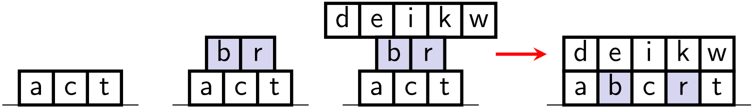
TimSort algorithm

a	c	t	r	b	w	k	i	e	d	u	n

- The input is split into **runs**, which are **monotonic subsequences**

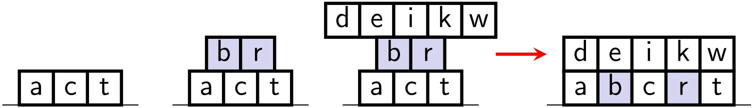
TimSort algorithm

- The input is split into **runs**, which are **monotonic subsequences**
- Every discovered run is added to a **stack**, then some **consecutive** runs can be **merged** (as in MergeSort) ← more details later



TimSort algorithm

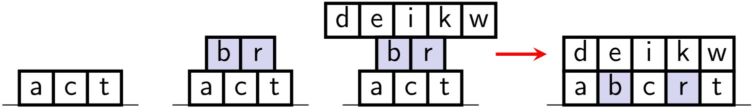
- The input is split into **runs**, which are **monotonic subsequences**
- Every discovered run is added to a **stack**, then some **consecutive** runs can be **merged** (as in MergeSort) ← more details later



- When there is no more run, the runs in the stack are merged top-down

TimSort algorithm

- The input is split into **runs**, which are **monotonic subsequences**
- Every discovered run is added to a **stack**, then some **consecutive** runs can be **merged** (as in MergeSort) ← more details later



- When there is no more run, the runs in the stack are merged top-down

Remark: TimSort also contains a lot of heuristics that we don't consider here (especially in the merge procedure)

timsort.txt (from Tim Peters)

```
 This describes an adaptive, stable, natural
mergesort, modestly called timsort (hey, I earned
it <wink>).  It has supernatural performance on many
kinds of partially ordered arrays (less than lg(N!)
comparisons needed, and as few as N-1), yet as fast
as Python's previous highly tuned samplesort hybrid on
random arrays.
```

```
 I believe that lists very often do have exploitable
partial order in real life, and this is the strongest
argument in favor of timsort
```

Running Time

In 2003, TimSort is announced to be in $\mathcal{O}(\log n!)$, with no formal proof.

Theorem (Auger, Nicaud, Pivoteau 2015)

TimSort has a **worst-case running time** of $\mathcal{O}(n \log n)$.

The proof is not very difficult, but hard to read (and to teach!)

Theorem (Folklore)

The running time of any sorting by comparisons algorithm is $\Omega(n \log n)$.

So TimSort is optimal, as many other algorithms: it does not explain why it is used in practice!

Parameterize Running Time

`I believe that lists very often do have exploitable partial order in real life, and this is the strongest argument in favor of timsort`

Idea: add a **parameter** to describe the running time

Parameterize Running Time

```
I believe that lists very often do have exploitable partial order in
real life, and this is the strongest argument in favor of timsort
```

Idea: add a **parameter** to describe the running time

- **First choice:** the **number of runs ρ**.
  It was conjectured that TimSort runs in $\mathcal{O}(n + n \log \rho)$

Parameterize Running Time

```
I believe that lists very often do have exploitable partial order in
real life, and this is the strongest argument in favor of timsort
```

Idea: add a **parameter** to describe the running time

- **First choice:** the **number of runs** ρ.
  It was conjectured that TimSort runs in $\mathcal{O}(n + n \log \rho)$
- **Better choice:** the **run lengths entropy** $\mathcal{H}$.
  If the runs have size $r_1, \ldots, r_\rho$, then

$$\mathcal{H} := -\sum_{i=1}^{\rho} \frac{r_i}{n} \log_2 \frac{r_i}{n}$$

Parameterize Running Time

```
I believe that lists very often do have exploitable partial order in
real life, and this is the strongest argument in favor of timsort
```

Idea: add a **parameter** to describe the running time

- **First choice:** the **number of runs** ρ.
  It was conjectured that TimSort runs in $\mathcal{O}(n + n \log \rho)$
- **Better choice:** the **run lengths entropy** $\mathcal{H}$.
  If the runs have size $r_1, \ldots, r_\rho$, then

$$\mathcal{H} := -\sum_{i=1}^{\rho} \frac{r_i}{n} \log_2 \frac{r_i}{n}$$

If the runs have sizes $\frac{n}{11}, \ldots \frac{n}{11}$: $\mathcal{H} = \log_2 11 \approx 3.46$
If the runs have sizes $\frac{90n}{100}, \frac{n}{100} \ldots \frac{n}{100}$: $\mathcal{H} \approx 0.80$
If the runs have sizes $\sqrt{n}, \ldots \sqrt{n}$: $\mathcal{H} = \frac{1}{2} \log_2 n$

Our results

Theorem (Auger, Jugé, Nicaud, Pivoteau. ESA 2018)

TimSort has a **worst-case running time** of $\mathcal{O}(n + n \log \rho)$.

Theorem (Auger, Jugé, Nicaud, Pivoteau. Talk ESA 2018)

TimSort has a **worst-case running time** of $\mathcal{O}(n + n\mathcal{H})$.

We always have $\mathcal{H} \leq \log_2 \rho \leq \log_2 n$.

Theorem (Auger, Jugé, Nicaud, Pivoteau. Buss, Knop 2019)

TimSort needs $1.5n\mathcal{H} + \mathcal{O}(n)$ comparisons in the worst case.

Theorem (Barbay, Navarro 2013)

Sorting by comparisons algorithms use more than $n\mathcal{H} - \mathcal{O}(n)$ comparisons.

Optimal algorithms?

Theorem (Auger, Jugé, Nicaud, Pivoteau. Buss, Knop 2019)

TimSort needs $1.5n\mathcal{H} + \mathcal{O}(n)$ comparisons in the worst case.

Theorem (Barbay, Navarro 2013)

Sorting by comparisons algorithms use more than $n\mathcal{H} - \mathcal{O}(n)$ comparisons.

There is a gap, and in fact, TimSort is not optimal (for this parameter $\mathcal{H}$).

Optimal algorithms?

Theorem (Auger, Jugé, Nicaud, Pivoteau. Buss, Knop 2019)

TimSort needs $1.5n\mathcal{H} + \mathcal{O}(n)$ comparisons in the worst case.

Theorem (Barbay, Navarro 2013)

Sorting by comparisons algorithms use more than $n\mathcal{H} - \mathcal{O}(n)$ comparisons.

There is a gap, and in fact, TimSort is not optimal (for this parameter $\mathcal{H}$).

Some optimal algorithms are known: **Takaoka 2009**, **Barbay & Navarro 2013**, **Munro & Wild 2018**.

So why analyzing TimSort?

Optimal algorithms?

Theorem (Auger, Jugé, Nicaud, Pivoteau. Buss, Knop 2019)

TimSort needs $1.5n\mathcal{H} + \mathcal{O}(n)$ comparisons in the worst case.

Theorem (Barbay, Navarro 2013)

Sorting by comparisons algorithms use more than $n\mathcal{H} - \mathcal{O}(n)$ comparisons.

There is a gap, and in fact, TimSort is not optimal (for this parameter $\mathcal{H}$).

Some optimal algorithms are known: **Takaoka 2009**, **Barbay & Navarro 2013**, **Munro & Wild 2018**.

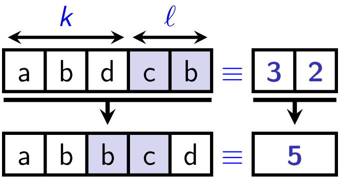
So why analyzing TimSort? because it is used in Python, Java, ...

Back to TimSort

Recall:

- monotonic runs are computed and added to a stack
- some merges of consecutive runs may happen when a run is added
- at the end, the remaining runs are merged top-down

Merges:

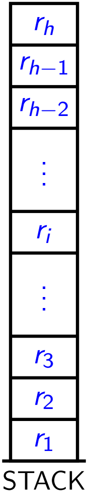


1. **Run merging** algorithm: standard + many optimizations
   - time $\mathcal{O}(k+\ell)$, using $k+\ell$ comparisons[1]
   - memory $\mathcal{O}(\min(k,\ell))$
2. **Policy** for choosing runs to merge:
   - depends on **run lengths** only

Let us forget array values – only remember run lengths!

[1] It is $k+\ell-1$, but we'll use $k+\ell$ to simplify.

TimSort's Merging Rules



Notations:

- the run R_i has length r_i
- the stack has height h
- the topmost run is R_h

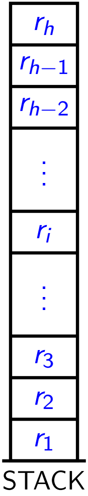
Merges after adding a new run:

- **While true**
  - **if** $r_h > r_{h-2}$ **then** merge R_{h-1} and R_{h-2}
  - **else if** $r_h \geq r_{h-1}$ **then** merge R_h and R_{h-1}
  - **else if** $r_h + r_{h-1} \geq r_{h-2}$ **then** merge R_h and R_{h-1}
  - **else** break

Remarks:

- we only consider the three topmost runs
- we only merge R_h and R_{h-1}, or R_{h-1} and R_{h-2}

TimSort's Merging Rules



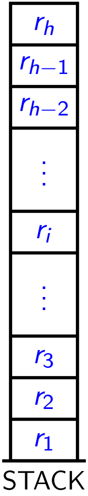
Merges after adding a new run:

- **While true**
  - **if** $r_h > r_{h-2}$ **then** merge R_{h-1} and R_{h-2}
  - **else if** $r_h \geq r_{h-1}$ **then** merge R_h and R_{h-1}
  - **else if** $r_h + r_{h-1} \geq r_{h-2}$ **then** merge R_h and R_{h-1}
  - **else** break

timsort.txt:

`Note that, by induction, it implies the lengths of pending runs form a decreasing sequence. It implies that, reading the lengths right to left, the pending-run lengths grow at least as fast as the Fibonacci numbers. Therefore the stack can never grow larger than about` $log_{\phi}(N)$ `entries`

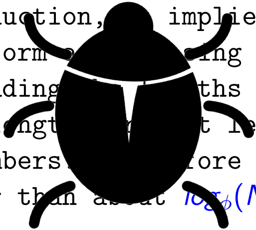
TimSort's Merging Rules



Merges after adding a new run:

- **While true**
  - **if** $r_h > r_{h-2}$ **then** merge R_{h-1} and R_{h-2}
  - **else if** $r_h \geq r_{h-1}$ **then** merge R_h and R_{h-1}
  - **else if** $r_h + r_{h-1} \geq r_{h-2}$ **then** merge R_h and R_{h-1}
  - **else** break

timsort.txt:



```
Note that, by induction, [illegible] implies the lengths
of pending runs form a decreasing sequence.  It
implies that, reading the lengths right to left,
the pending-run lengths grow at least as fast as
the Fibonacci numbers.  Therefore the stack can
never grow larger than about log_φ(N) entries
```

An error in timsort.txt

- **While true**
  - **if** $r_h > r_{h-2}$ **then** merge R_{h-1} and R_{h-2}
  - **else if** $r_h \geq r_{h-1}$ **then** merge R_h and R_{h-1}
  - **else if** $r_h + r_{h-1} \geq r_{h-2}$ **then** merge R_h and R_{h-1}
  - **else** break

The invariant $r_{i+2} + r_{i+1} < r_i$ does not hold!
Discovered by de Gouw et al (2015) while trying to prove (formally) the correctness of Java's Timsort, using KeY (verification tool for Java)

An error in timsort.txt

- **While true**
  - **if** $r_h > r_{h-2}$ **then** merge R_{h-1} and R_{h-2}
  - **else if** $r_h \geq r_{h-1}$ **then** merge R_h and R_{h-1}
  - **else if** $r_h + r_{h-1} \geq r_{h-2}$ **then** merge R_h and R_{h-1}
  - **else** break

The invariant $r_{i+2} + r_{i+1} < r_i$ does not hold!
Discovered by de Gouw et al (2015) while trying to prove (formally) the correctness of Java's Timsort, using KeY (verification tool for Java)

Is it a real problem?

- In **Python**: not really, the algorithm is still efficient and correct
- In **Java**: they use the invariant to fix the maximum size of the stack, implemented with a static array $\Rightarrow$ de Gouw et al (2015) built an array that produces an error for Java's sort()!

Two versions of TimSort

de Gouw et al (2015) proposed two solutions to fix the problem:

1. Adding a new rule (implemented in Python)
   - **While true**
     - **if** $r_h > r_{h-2}$ **then** merge R_{h-1} and R_{h-2}
     - **else if** $r_h \geq r_{h-1}$ **then** merge R_h and R_{h-1}
     - **else if** $r_h + r_{h-1} \geq r_{h-2}$ **then** merge R_h and R_{h-1}
     - **else if** $r_{h-1} + r_{h-2} \geq r_{h-3}$ **then** merge R_h and R_{h-1}
     - **else** break

   The invariant now holds, the algorithm is certified in KeY.

2. Computing correct maximal heights for the stack (implemented in Java)

Lemma

Throughout execution of TimSort, the invariant cannot be violated at two consecutive runs in the stack.

Running time analysis: $\mathcal{O}(n \log n)$

We focus on the main loop: other parts are done in $\mathcal{O}(n)$ comparisons.

- **While there are remaining runs**

  (#1) Add a new run to the stack
  Repeat until stabilized

  (#2) **if** $r_h > r_{h-2}$ **then** merge R_{h-1} and R_{h-2}
  (#3) **else if** $r_h \geq r_{h-1}$ **then** merge R_h and R_{h-1}
  (#4) **else if** $r_h + r_{h-1} \geq r_{h-2}$ **then** merge R_h and R_{h-1}
  (#5) **else if** $r_{h-1} + r_{h-2} \geq r_{h-3}$ **then** merge R_h and R_{h-1}

Amortized analysis:

- $\diamondsuit$-tokens and $\heartsuit$-tokens are given to the elements of the input
- tokens are used to pay for comparisons
- the total number of tokens granted is our upper bound

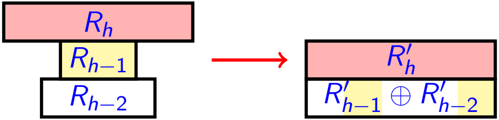
Tokens' rules: an element gets two $\diamondsuit$ and one $\heartsuit$

- when its run enters the stack
- when its height in the stack decreases

Running time analysis: $\mathcal{O}(n \log n)$, case #2

(#2) **if** $r_h > r_{h-2}$ **then** merge R_{h-1} and R_{h-2}

Every element of R_h and R_{h-1} pays one $\diamondsuit$: the merge cost is $r_{h-1} + r_{h-2} \leq r_{h-1} + r_h$, hence it is fully paid.

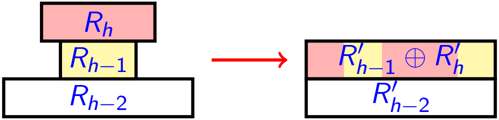


The height of every element that paid one $\diamondsuit$ decreases by one: they all gain two $\diamondsuit$ and one $\heartsuit$

Running time analysis: $\mathcal{O}(n \log n)$, case #3

(#3) **else if** $r_h \geq r_{h-1}$ **then** merge R_h and R_{h-1}

Every element of R_h pays two $\diamondsuit$: the merge cost is $r_h + r_{h-1} \leq 2r_h$, hence it is fully paid.

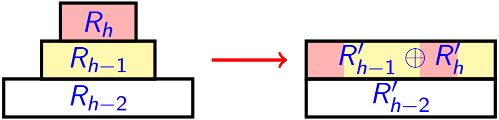


The height of every element that paid two $\diamondsuit$ decreases by one: they all gain two $\diamondsuit$ and one $\heartsuit$

Running time analysis: $\mathcal{O}(n \log n)$, case #4

(#4) **else if** $r_h + r_{h-1} \geq r_{h-2}$ **then** merge R_h and R_{h-1}

Every element of R_h pays one $\diamondsuit$, every element of R_{h-1} pays one $\heartsuit$: the merge cost is $r_h + r_{h-1}$, hence it is fully paid.



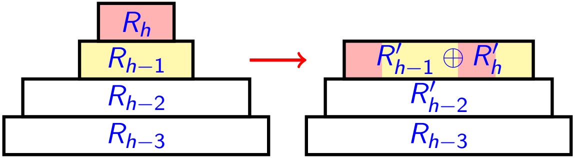
The height of the elements of R_h decreases by one: **ok** for $\diamondsuit$

- Elements that paid one $\heartsuit$ are now in the topmost run
- Elements in the topmost run never pay with $\heartsuit$
- In the new stack, $r_h \geq r_{h-1}$ so another merge is going to occur (#3)
- The height of the new topmost run is going to decrease during this new merge, its elements will get two $\diamondsuit$ and one $\heartsuit$

Running time analysis: $\mathcal{O}(n \log n)$, case #5

(#5) **else if** $r_{h-1} + r_{h-2} \geq r_{h-3}$ **then** merge R_h and R_{h-1}

Every element of R_h pays one $\diamondsuit$, every element of R_{h-1} pays one $\heartsuit$: the merge cost is $r_h + r_{h-1}$, hence it is fully paid.



The height of the elements of R_h decreases by one: **ok** for $\diamondsuit$

- Elements that paid one $\heartsuit$ are now in the topmost run
- Elements in the topmost run never pay with $\heartsuit$
- In the new stack, $r_h + r_{h-1} \geq r_{h-2}$ so another merge is going to occur (#4)
- The height of the new topmost run is going to decrease during this new merge, its elements will get two $\diamondsuit$ and one $\heartsuit$

Running time analysis: $\mathcal{O}(n \log n)$

Summary:

- Computing the run decomposition takes $\mathcal{O}(n)$
- For the main loop:
  - each element gets $2\diamondsuit$ and $1\heartsuit$ when entering the stack
  - each merge is paid with $\diamondsuit$ and $\heartsuit$
  - when an element pays with $\diamondsuit$, it get it (them) back immediately after
  - when an element pays with $\heartsuit$, another merge occurs just after, during which it get it back
- The final merges are done in $\mathcal{O}(n)$ by direct computation

Lemma

At any moment during TimSort, the stack has height in $\mathcal{O}(\log n)$.

Proof: the invariant holds.

Theorem (Auger, Jugé, Nicaud, Pivoteau 2018)

The running time of TimSort is in $\mathcal{O}(n \log n)$.

Running time analysis: $\mathcal{O}(n + n\mathcal{H})$

Recall: #1 is the insertion of a new run in the stack
Recall: $\mathcal{H} = -\sum \frac{r_i}{n} \log \frac{r_i}{n}$

We use the following decomposition of the sequence of events:

#1 #2 #2 #2	#3 #4 #2 #5 #3	#1 #2 #2	#4 #2 #2 #3
starting sequence pay with ♠	ending sequence pay with ♢and ♡	starting sequence pay with ♠	ending sequence pay with ♢and ♡

Two lemmas (both consequences of the invariant):

- The total cost in ♠-tokens is linear
- The height of the stack at the beginning of the ending sequence after inserting a run of length r is $\mathcal{O}(\log \frac{n}{r})$.

Theorem (Auger, Jugé, Nicaud, Pivoteau 2018)

The running time of TimSort is in $\mathcal{O}(n + n\mathcal{H})$.

Running time analysis: summary

We proved that for the "new" TimSort, we have:

Theorem (Auger, Jugé, Nicaud, Pivoteau 2018)

The running time of TimSort is in $\mathcal{O}(n + n\mathcal{H})$.

This can be improved to:

Theorem (Auger, Jugé, Nicaud, Pivoteau 2019)

TimSort needs at most $1.5n\mathcal{H} + \mathcal{O}(n)$ comparisons.

Running time analysis: summary

We proved that for the "new" TimSort, we have:

Theorem (Auger, Jugé, Nicaud, Pivoteau 2018)

The running time of TimSort is in $\mathcal{O}(n + n\mathcal{H})$.

This can be improved to:

Theorem (Auger, Jugé, Nicaud, Pivoteau 2019)

TimSort needs at most $1.5n\mathcal{H} + \mathcal{O}(n)$ comparisons.

What about the Java's version of TimSort? it is also in $\mathcal{O}(n + n \log \rho)$, but it is much more complicated to establish (no nice invariant).

Running time analysis: summary

We proved that for the "new" TimSort, we have:

Theorem (Auger, Jugé, Nicaud, Pivoteau 2018)

The running time of TimSort is in $\mathcal{O}(n + n\mathcal{H})$.

This can be improved to:

Theorem (Auger, Jugé, Nicaud, Pivoteau 2019)

TimSort needs at most $1.5n\mathcal{H} + \mathcal{O}(n)$ comparisons.

What about the Java's version of TimSort? it is also in $\mathcal{O}(n + n\log\rho)$, but it is much more complicated to establish (no nice invariant).

but wait a minute ...

Another bug in Java's TimSort

Lemma

Throughout execution of TimSort, the invariant cannot be violated at two consecutive runs in the stack.

Another bug in Java's TimSort

Lemma

Throughout execution of TimSort, the invariant cannot be violated at two consecutive runs in the stack.

The lemma is incorrect!

We built an array that produces an error to Java's (patched) TimSort!

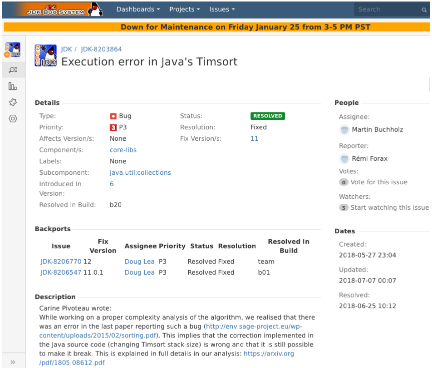
Another bug in Java's TimSort

Lemma

Throughout execution of TimSort, the invariant cannot be violated at two consecutive runs in the stack.

The lemma is incorrect!

We built an array that produces an error to Java's (patched) TimSort!



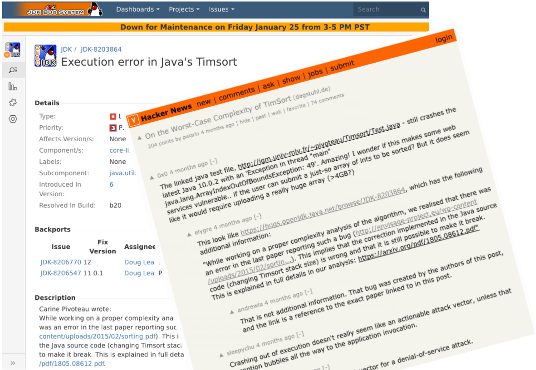
Another bug in Java's TimSort

Lemma

Throughout execution of TimSort, the invariant cannot be violated at two consecutive runs in the stack.

The lemma is incorrect!

We built an array that produces an error to Java's (patched) TimSort!



Another bug in Java's TimSort

Lemma

Throughout execution of TimSort, the invariant cannot be violated at two consecutive runs in the stack.

The lemma is incorrect!

We built an array that produces an error to Java's (patched) TimSort!

Conclusion for TimSort

- TimSort is an efficient algorithm, in theory and in practice
- It is not entropy-optimal, but not far from it
- There are many optimisation to build the runs, to perform the merges, . . .

- Its $\mathcal{O}(n \log n)$ running time was proved more than 10 years after it was announced
- There were two consecutive bugs in Java's version, due to improper analysis of the algorithm

Conclusion for TimSort

- TimSort is an efficient algorithm, in theory and in practice
- It is not entropy-optimal, but not far from it
- There are many optimisation to build the runs, to perform the merges, ...

- Its $\mathcal{O}(n \log n)$ running time was proved more than 10 years after it was announced
- There were two consecutive bugs in Java's version, due to improper analysis of the algorithm

Conclusion for TimSort

- TimSort is an efficient algorithm, in theory and in practice
- It is not entropy-optimal, but not far from it
- There are many optimisation to build the runs, to perform the merges, ...

- Its $\mathcal{O}(n \log n)$ running time was proved more than 10 years after it was announced
- There were two consecutive bugs in Java's version, due to improper analysis of the algorithm

Every used algorithm deserves a **fine grain analysis**

II. Branch predictions

with N. Auger & C. Pivoteau

A toy example: looking for the min and the max

We want to find the minimum and the maximum of an array T of size n.

```
min = T[n-1];
max = T[n-1];
for(i=0; i<n-1; i++){
    a = T[i];
    if (a < min) min = a;
    if (a > max) max = a;
}
```

Naive solution:
foreach element a of T, if a is smaller than the current minimum, update the minimum; if it is greater than the current maximum, update the maximum.

Fact: the naive solution uses $2n - 2 \sim 2n$ comparisons.

Can we do better?

Min & Max: Optimal Algorithm

Idea: take the elements by pairs (a_1, a_2), compare them, then compare the smallest to the current min and the largest to the current max

```
min = max = T[n-1];
for(i=0; i<n-1; i+=2){
    a1 = T[i];
    a2 = T[i+1];
    if (a1 < a2) {
      if (a1 < min) min = a1;
      if (a2 > max) max = a2;
    }
    else {
      if (a2 < min) min = a2;
      if (a1 > max) max = a1;
    }
}
```

Number of comparisons:

- $\sim \frac{n}{2}$ loop iterations
- 3 comparisons by iterations
- number of comparisons: $\sim \frac{3}{2}n$

That's better!

Min & Max: Optimal Algorithm

Idea: take the elements by pairs (a_1, a_2), compare them, then compare the smallest to the current min and the largest to the current max

```
min = max = T[n-1];
for(i=0; i<n-1; i+=2){
    a1 = T[i];
    a2 = T[i+1];
    if (a1 < a2) {
      if (a1 < min) min = a1;
      if (a2 > max) max = a2;
    }
    else {
      if (a2 < min) min = a2;
      if (a1 > max) max = a1;
    }
}
```

Number of comparisons:

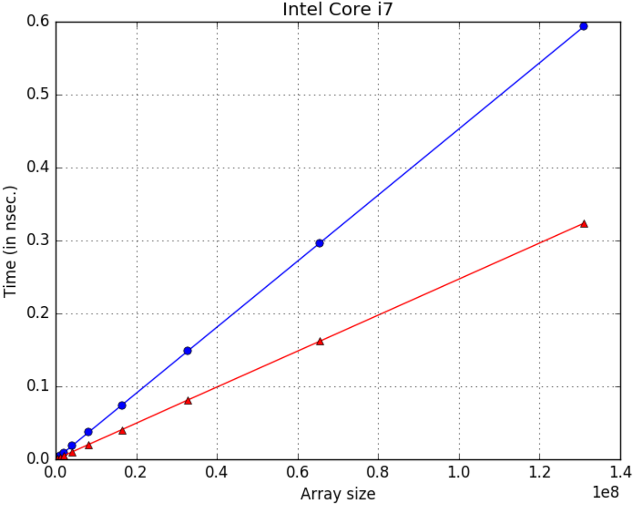
- $\sim \frac{n}{2}$ loop iterations
- 3 comparisons by iterations
- number of comparisons: $\sim \frac{3}{2}n$

That's better!

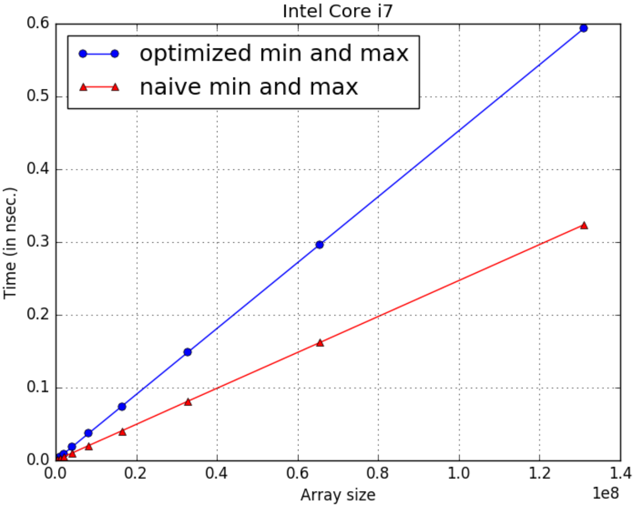
Theorem (Folklore)

At least $\sim \frac{3}{2}n$ comparisons are needed to compute the min and the max.

Min & Max: experiments



Min & Max: experiments



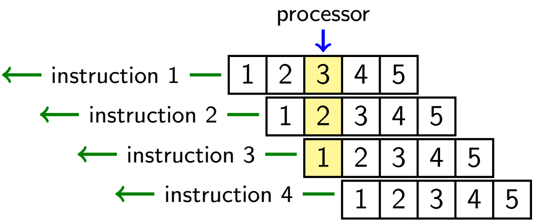
The naive solution is more efficient in practice!

Pipeline

Notion of pipeline:

- During the execution of a program, instructions are executed sequentially: `i=3`, `a<b`, `if (...)`, ...
- Instructions are divided into several sequential steps
- Different steps can be handle in parallel by the processor

Example with 5 steps:

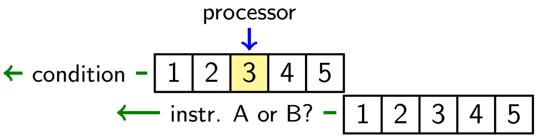


It can be up to five time as fast

Pipeline and Branches

- A **branch** is an instruction with several possible following instructions: `if`, `while`, . . .
- Branches constitute a problem for the pipeline:

```
if <condition>
   <instruction A>;
else
   <instruction B>;
```

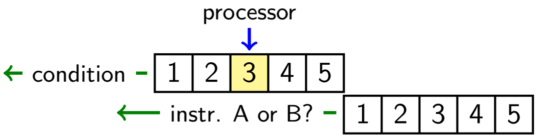


- We have to wait for the completion of all the stages of `<condition>` to know whether it is followed by `A` or by `B`!

Pipeline and Branches

- A **branch** is an instruction with several possible following instructions: `if`, `while`, ...
- Branches constitute a problem for the pipeline:

```
if <condition>
   <instruction A>;
else
   <instruction B>;
```



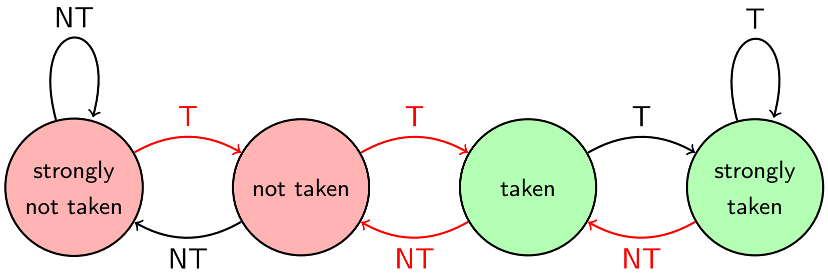
- We have to wait for the completion of all the stages of `<condition>` to know whether it is followed by `A` or by `B`!

Solution: try to anticipate if condition = **true** or **false**

Branch predictions

- Branches does not fit well with the pipeline
- We try to anticipate whether the branch will be:
  - **Taken (T)**: when `<condition>` is **true**
  - **Not Taken (NT)**: when `<condition>` is **false**
- We push the predicted next instruction in the pipeline:
  - if the prediction is **correct**, we **gain** some time
  - if it is **incorrect**, we have to undo what we did, we **lose** some time

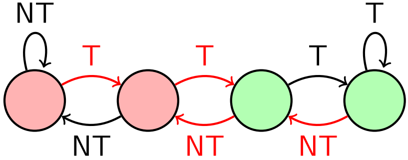
A simple **local predictor**, the **2-bit predictor** (one for each branch):



Back to the toy example

```
min = T[n-1];
max = T[n-1];
for(i=0; i<n-1; i++){
    a = T[i];
    if (a < min) min = a;
    if (a > max) max = a;
}
```

The condition `if (a < min)` is true when there is a **min-record**, and false otherwise.



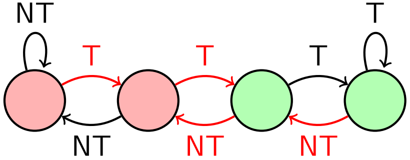
We have a pure AofA exercise:

- Start at any state, draw a uniform random permutation
- Scan it from left to right: when there is a min-record, go to the right in the automaton (if possible), otherwise go to the left
- What is the **expected number of mispredictions**?

Back to the toy example

```
min = T[n-1];
max = T[n-1];
for(i=0; i<n-1; i++){
    a = T[i];
    if (a < min) min = a;
    if (a > max) max = a;
}
```

The condition `if (a < min)` is true when there is a **min-record**, and false otherwise.



We have a pure AofA exercise:

- Start at any state, draw a uniform random permutation
- Scan it from left to right: when there is a min-record, go to the right in the automaton (if possible), otherwise go to the left
- What is the **expected number of mispredictions**?

Lemma

The expected number of mispredictions produced by each `if` in the naive solution is asymptotically equivalent to $\log n$.

What About the Optimal Algorithm?

Idea: take the elements by pairs (a_1, a_2), compare them, then compare the smallest to the current min and the largest to the current max

```
min = max = T[n-1];
for(i=0; i<n-1; i+=2){
    a1 = T[i];
    a2 = T[i+1];
    if (a1 < a2) {
      if (a1 < min) min = a1;
      if (a2 > max) max = a2;
    }
    else {
      if (a2 < min) min = a2;
      if (a1 > max) max = a1;
    }
}
```

- The first branch: `if (a1 < a2)` is **true** with probability $\frac{1}{2}$ for uniform random permutations.
- This cannot be well predicted: there is a misprediction here with probability $\frac{1}{2}$ for each loop iteration
- The expected number of mispredictions is asymptotically $\frac{n}{4}$!

Toy example: conclusion

- We proposed to solutions to this simple problem
- For uniform random permutations, in expectation:
  - The naive algorithm uses $2n$ comparisons and $2\log n$ mispredictions
  - The optimal algorithm uses $\frac{3}{2}n$ comparisons and $\frac{1}{4}n$ mispredictions
  - Experimentally, the naive algorithm is more efficient!

Toy example: conclusion

- We proposed to solutions to this simple problem
- For uniform random permutations, in expectation:
  - The naive algorithm uses $2n$ comparisons and $2\log n$ mispredictions
  - The optimal algorithm uses $\frac{3}{2}n$ comparisons and $\frac{1}{4}n$ mispredictions
  - Experimentally, the naive algorithm is more efficient!

We have to add branch predictors to our model of computation (RAM model) to fully describe the complexity of some algorithms.

Some Related Works

- 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

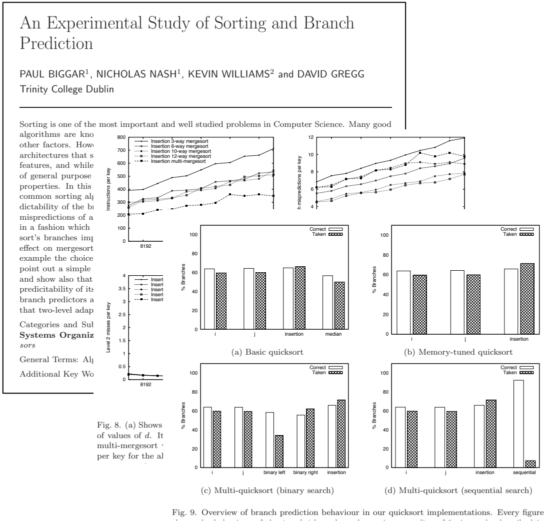
Abstract. Branch mispredictions is an important factor affecting the running time in practice. In this paper we consider tradeoffs between the number of branch mispredictions and the number of comparisons for
sorting
algorith
mispred
by adop
tions. F
rithm p
$\Omega(n \log$
of inver
by Esti
col and
mispred

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

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

Some Related Works

- 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 br mispredictions of a in a fashion which sort's branches imp effect on mergesort example the choice point out a simple and show also that predictability of its branch predictors a that two-level adap

Categories and Sub **Systems Organiz** *sors*

General Terms: Alg

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

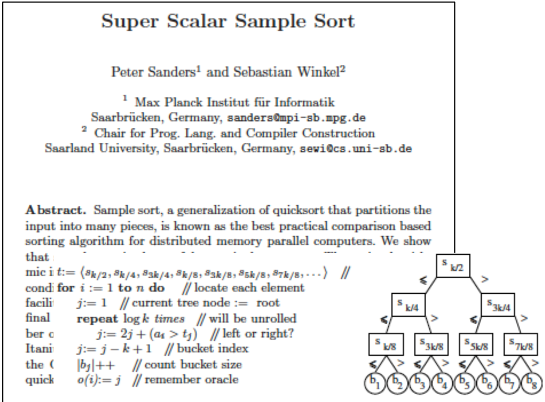
(c) Multi-quicksort (binary search)

(d) Multi-quicksort (sequential search)

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

Some Related Works

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

Some Related Works

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

Some Related Works

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

How Branch Mispredictions Affect Quicksort

Kanela Kaligosi[1] and Peter Sanders[2]

[1] Max Planck Ir
Saarbrüc
kaligosi@
[2] Universität I
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	α-skewed pivot
static predictor	$\frac{\ln 2}{2} n \lg n + \mathcal{O}(n),\ \frac{\ln 2}{2} \approx 0.3466$	$\frac{\alpha}{H(\alpha)} n \lg n + \mathcal{O}(n),\ \alpha < 1/2$ $\frac{1-\alpha}{H(\alpha)} n \lg n + \mathcal{O}(n),\ \alpha \geq 1/2$
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	$\frac{28 \ln 2}{45} n \lg n + \mathcal{O}(n),\ \frac{28 \ln 2}{45} \approx 0.4313$	$\frac{2\alpha^4 - 4\alpha^3 + \alpha^2 + \alpha}{(1-\alpha(1-\alpha))H(\alpha)} n \lg n + \mathcal{O}(n)$

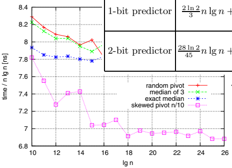


Fig. 3. Time / $n \lg n$ for random pivot, median of 3, exact median, 1/10-skewed pivot

Some Related Works

- Brodal & Moruz, 2005 : mispredictions and (adaptive) sorting
- Biggar et al, 2008 : experimental, branch prediction and sorting
- Sanders and Winkel, 2004 : quicksort variant without branches
- Elmasry et al, 2012 : mergesort variant without branches
- Kaligosi and 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 tim… get a reliab… sort: It has… under certa… sically opti… dian of a sa… 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.

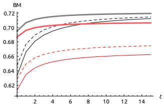


Figure 5: Branch mispredictions, as a function of t, in CQS (black) and YQS (red) with 1-bit branch prediction (fat), 2-bit saturating counter (thin solid) and 2-bit flip-consecutive (dashed) using symmetric sampling: $\mathbf{t}_{\mathrm{CQS}} = (3t+2, 3t+2)$ and $\mathbf{t}_{\mathrm{YQS}} = (2t+1,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

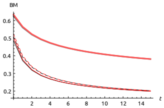
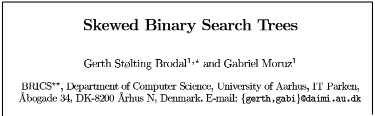


Figure 6: Branch mispredictions, as a function of t, in CQS (black) and YQS (red) with 1-bit (fat), 2-bit sc (thin solid) and 2-bit fc (dashed) predictors, using extremely skewed sampling: $\mathbf{t}_{\mathrm{CQS}} = (0, 6t+4)$ and $\mathbf{t}_{\mathrm{YQS}} = (0, 6t+3, 0)$

Some Related Works

- Brodal & Moruz, 2005 : mispredictions and (adaptive) sorting
- Biggar et al, 2008 : experimental, branch prediction and sorting
- Sanders and Winkel, 2004 : quicksort variant without branches
- Elmasry et al, 2012 : mergesort variant without branches
- Kaligosi and 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 cons
trees). In this paper we prese
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%.

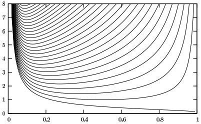


Fig. 1. Bound on the expected cost for a random search, where the cost for visiting the left child is $c_l = 1$ and the cost for processing the right child is $c_r = 0, 1, 2, \ldots, 28$ ($c_r = 0$ being the lowest curve).

Some Related Works

- Brodal & Moruz, 2005 : mispredictions and (adaptive) sorting
- Biggar et al, 2008 : experimental, branch prediction and sorting
- Sanders and Winkel, 2004 : quicksort variant without branches
- Elmasry et al, 2012 : mergesort variant without branches
- Kaligosi and 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 cons
trees). In this paper we prese
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%.

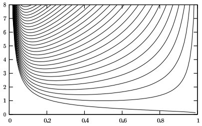


Fig. 1. Bound on the expected cost for a random search, where the cost for visiting the left child is $c_l = 1$ and the cost for processing the right child is $c_r = 0, 1, 2, \ldots, 28$ ($c_r = 0$ being the lowest curve).

What next?

- Branch predictors **exist** in computers
- They cannot easily be turned off
- **Classical paradigm:** ignore them, they are doing their job
- **AofA:** sometimes, it is necessary to take them into account

*What if we take them into account to **design** new algorithms?*

Exponentiation by Squaring

We consider the classical **Exponentiation by Squaring** algorithm, and we unroll the main loop, to have two iterations each time.

pow(x,n)

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

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

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;
}
```

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

Exponentiation by Squaring

If n is taken uniformly at random in $\{0, \ldots, 4^k\}$, then each `if` is taken with probability $\frac{1}{2}$: it is difficult to predict.

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;
}
```

Exponentiation by Squaring

Idea: guide the predictors using a **unnecessary** test!

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;
}
```

guided(x,n)

```
r = 1;
while (n > 0) {
  t = x * x;
  // n0 n1 ≠ 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;
}
```

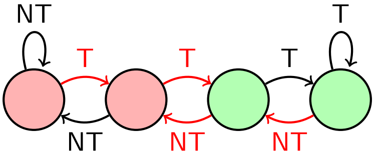
We have one more comparison by iteration, but predictions are easier.

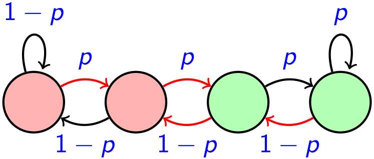
Exponentiation by Squaring

Results:

- **25 % more comparisons** for guided than for unrolled
- guided is 14% faster than unrolled
- yet, the number of multiplications is essentially the **same**.

Analysis: Markov chains!

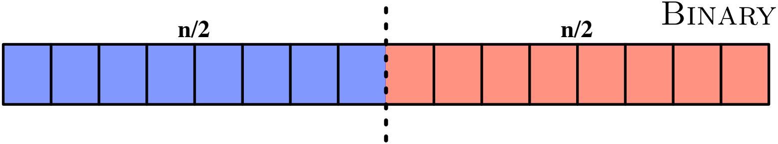




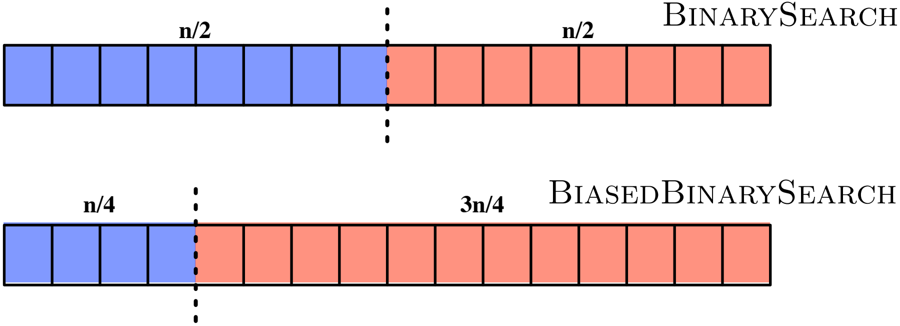
- The expected number of mispredictions after k steps in the Markov chain is asymptotically $\mu(p)k$, with $\mu(p) = \frac{p(1-p)}{1-2p(1-p)}$.
- The expected number of mispredictions in guided is $\alpha \log_2 n$, with $\alpha = \frac{1}{2}\mu(3/4) + \frac{3}{4}\mu(2/3) = 0.45$

Binary Search

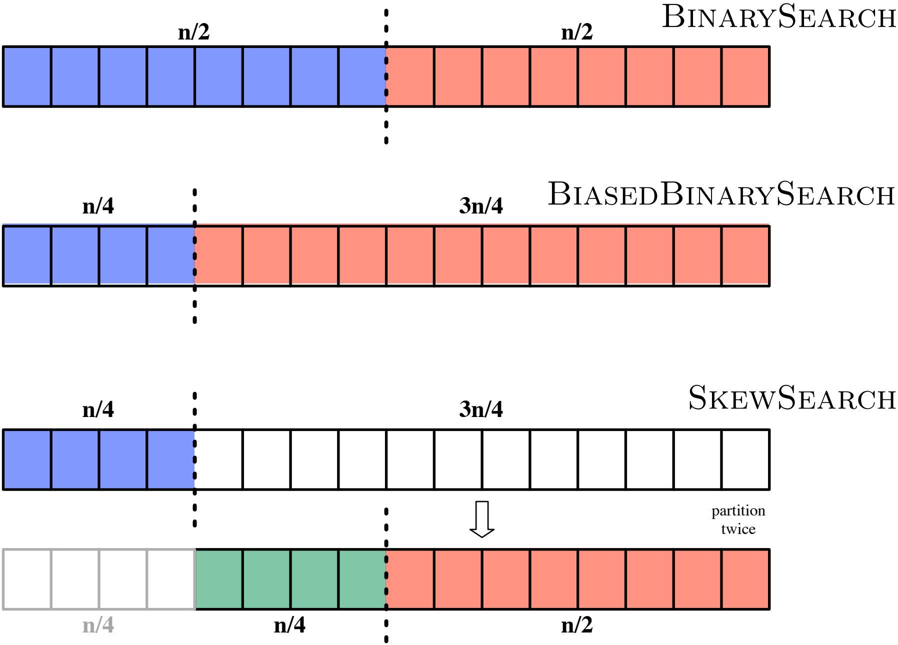
BinarySearch



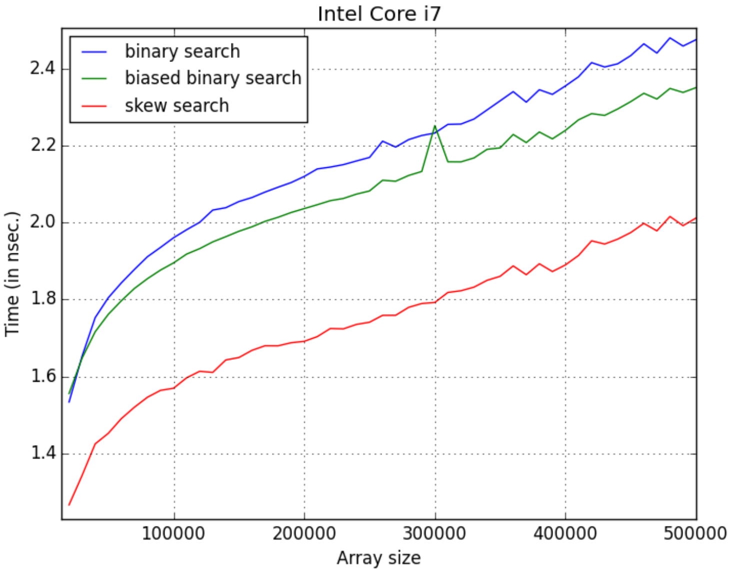
Binary Search



Binary Search



Binary Search



Binary Search: Analysis

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

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

Proof:

- Master Theorem gives the expected number of times each conditional is executed
- Ensure that our predictors behave *almost* like Markov chains.

Branch Predictions: Conclusion

- Branch prediction mechanism alters the running time of algorithms
- It explains why the naive solution is better for the min/max problem
- We use it to **finely tune** classical algorithms:
  - Exponentiation by squaring is more efficient by **adding a useless test!**
  - Binary search is more efficient if we **don't cut in the middle!**

- The importance of branch mispredictions is limited to basic algorithms

Thank you!