Quicksorts of the 21st century

Sebastian Wild

based on joint work with Stefan Edelkamp, Conrado Martínez, Markus Nebel, Ralph Neininger, and Armin Weiß

UNIVERSITY OF LIVERPOOL

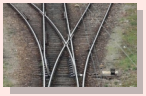
Isaac Newton Institute – Verified Software Workshop



Outline



1 Context



2 Dual-Pivot Quicksort



3 Average-Case Analysis



4 Quicksort + X

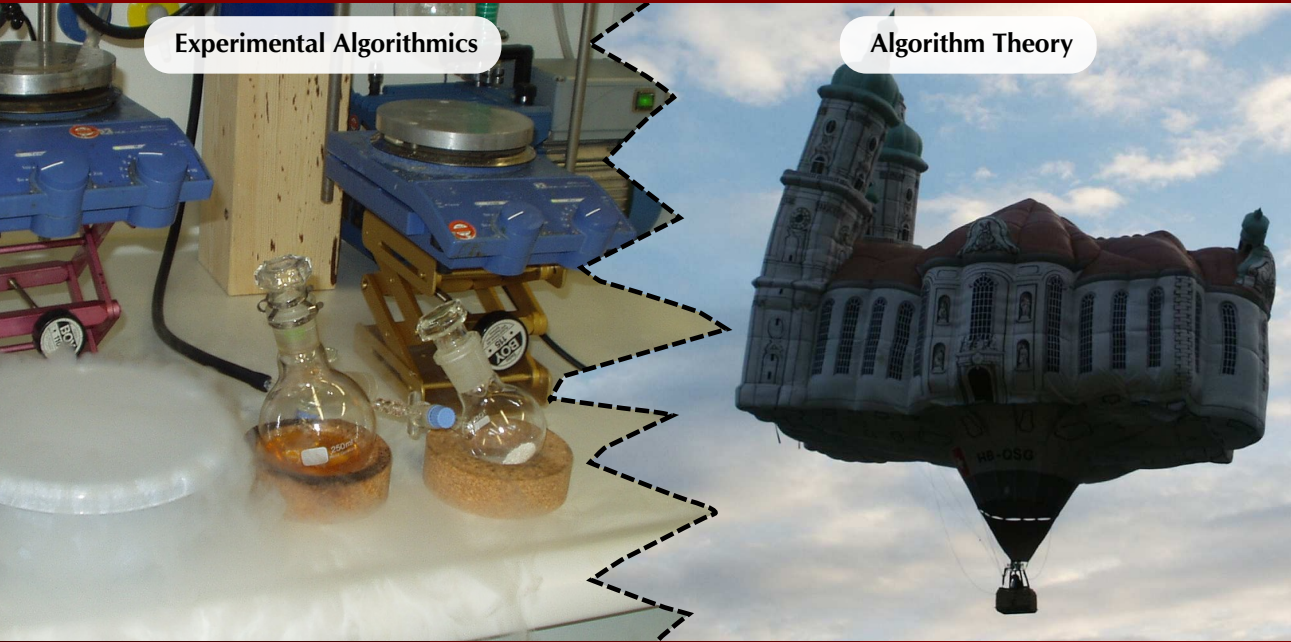


5 Conclusion

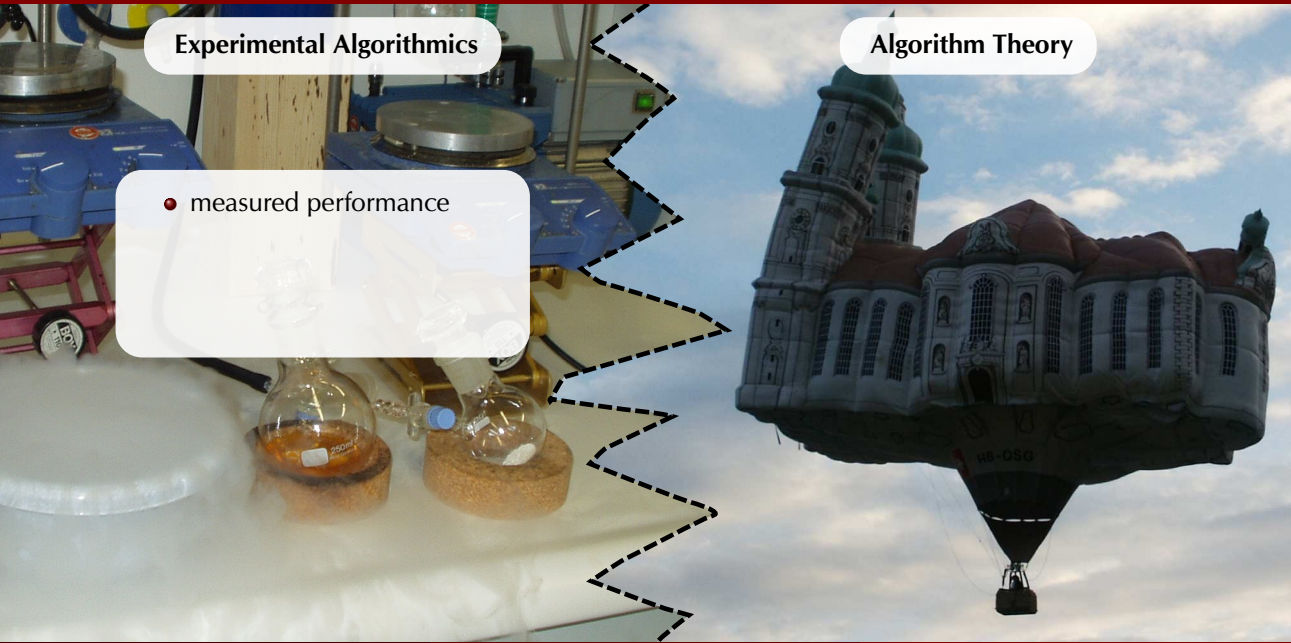


1 Context

Algorithm Science



Algorithm Science



Experimental Algorithmics

- measured performance

Algorithm Theory

Algorithm Science

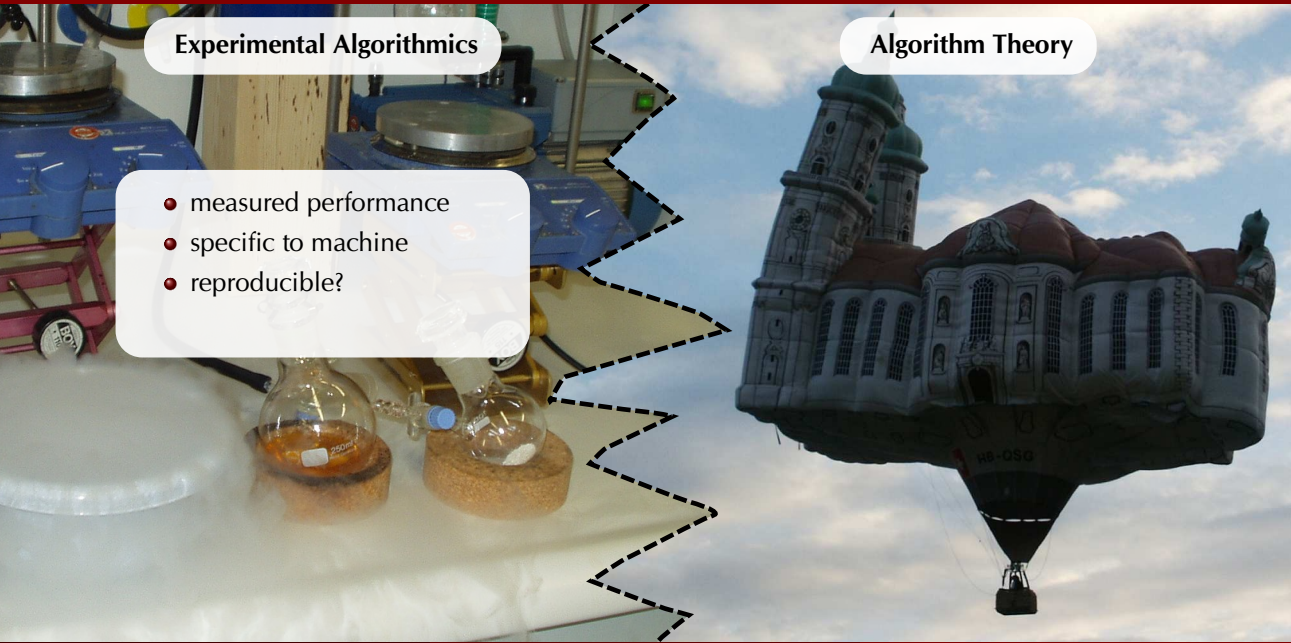
Experimental Algorithmics

- measured performance
- specific to machine

Algorithm Theory



Algorithm Science

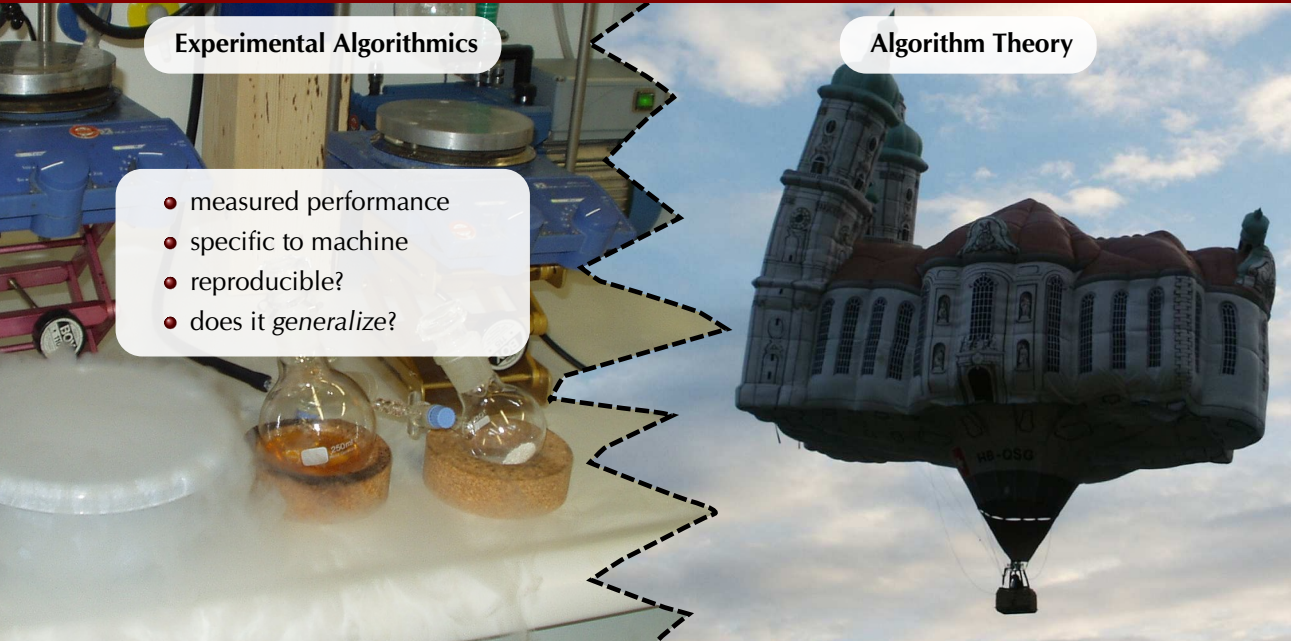


Experimental Algorithmics

- measured performance
- specific to machine
- reproducible?

Algorithm Theory

Algorithm Science

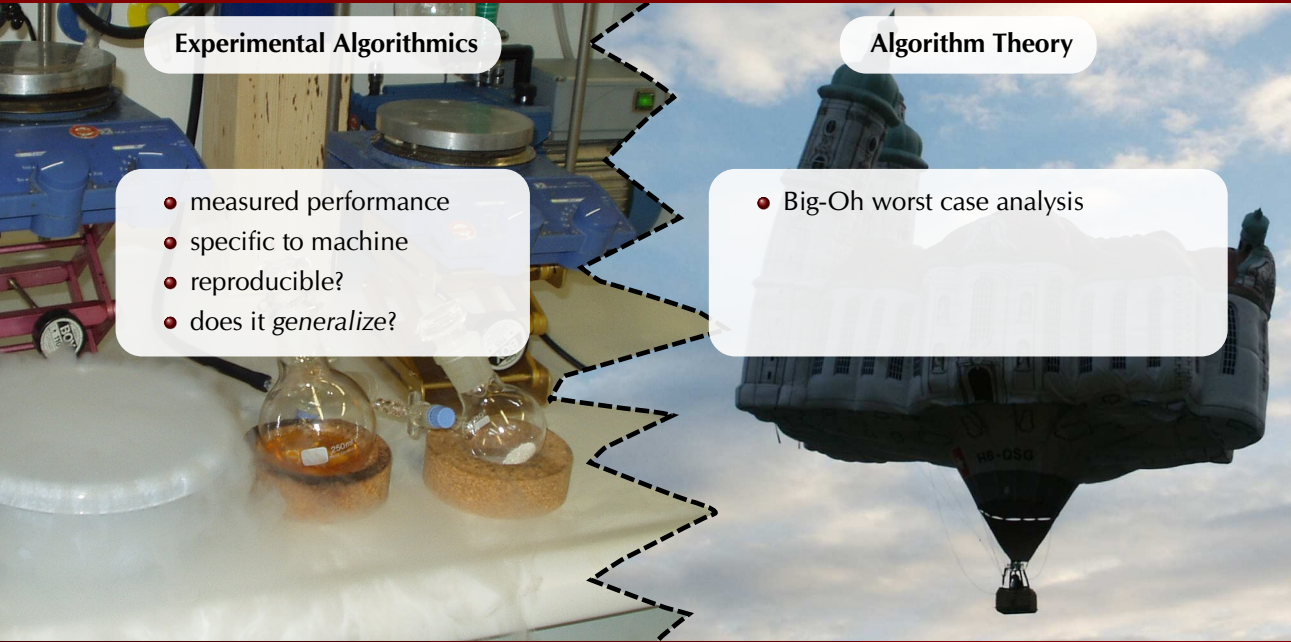


Experimental Algorithmics

- measured performance
- specific to machine
- reproducible?
- does it *generalize*?

Algorithm Theory

Algorithm Science



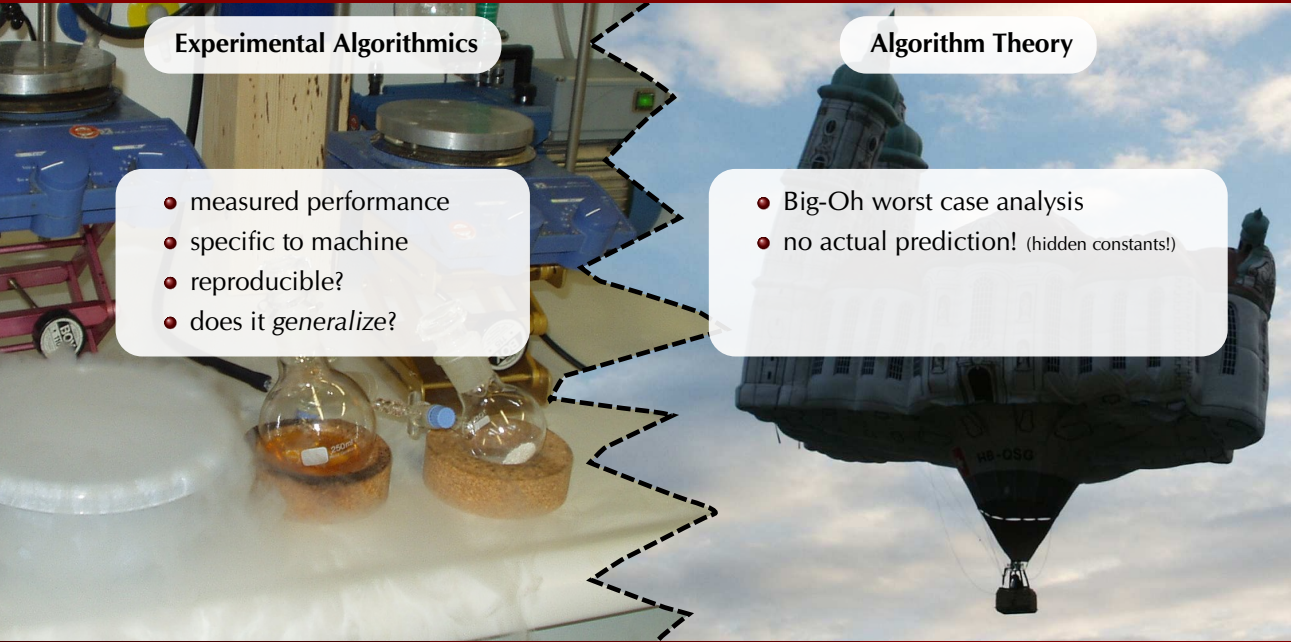
Experimental Algorithmics

- measured performance
- specific to machine
- reproducible?
- does it *generalize*?

Algorithm Theory

- Big-Oh worst case analysis

Algorithm Science



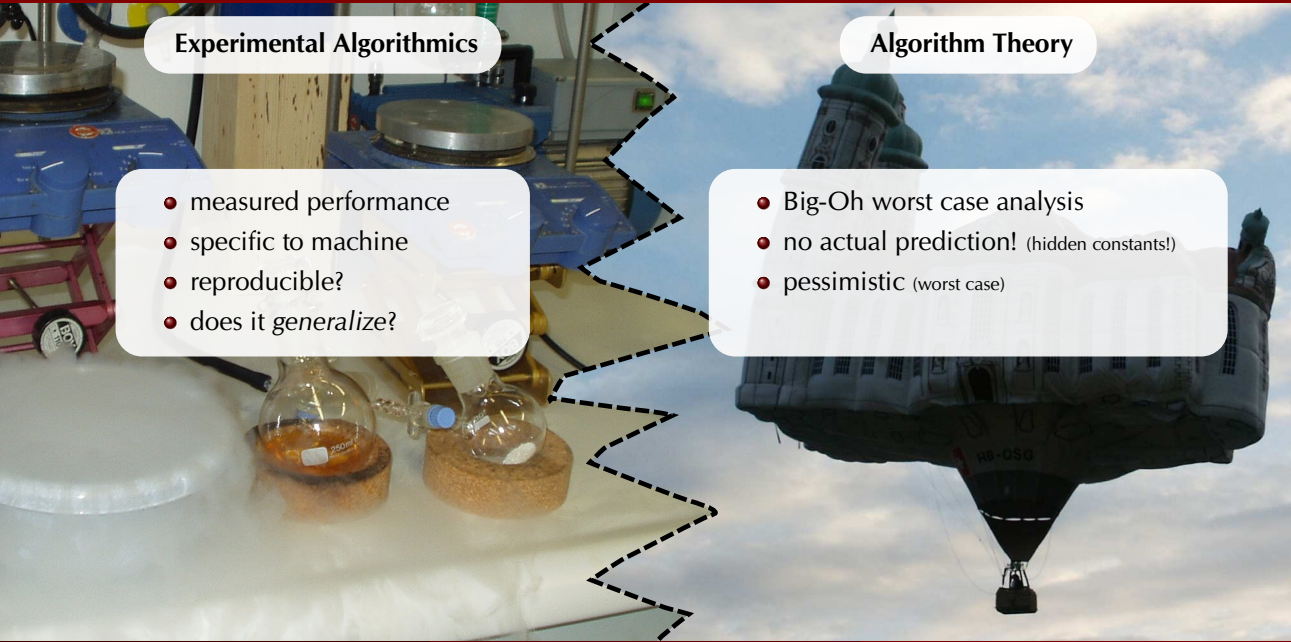
Experimental Algorithmics

- measured performance
- specific to machine
- reproducible?
- does it *generalize*?

Algorithm Theory

- Big-Oh worst case analysis
- no actual prediction! (hidden constants!)

Algorithm Science



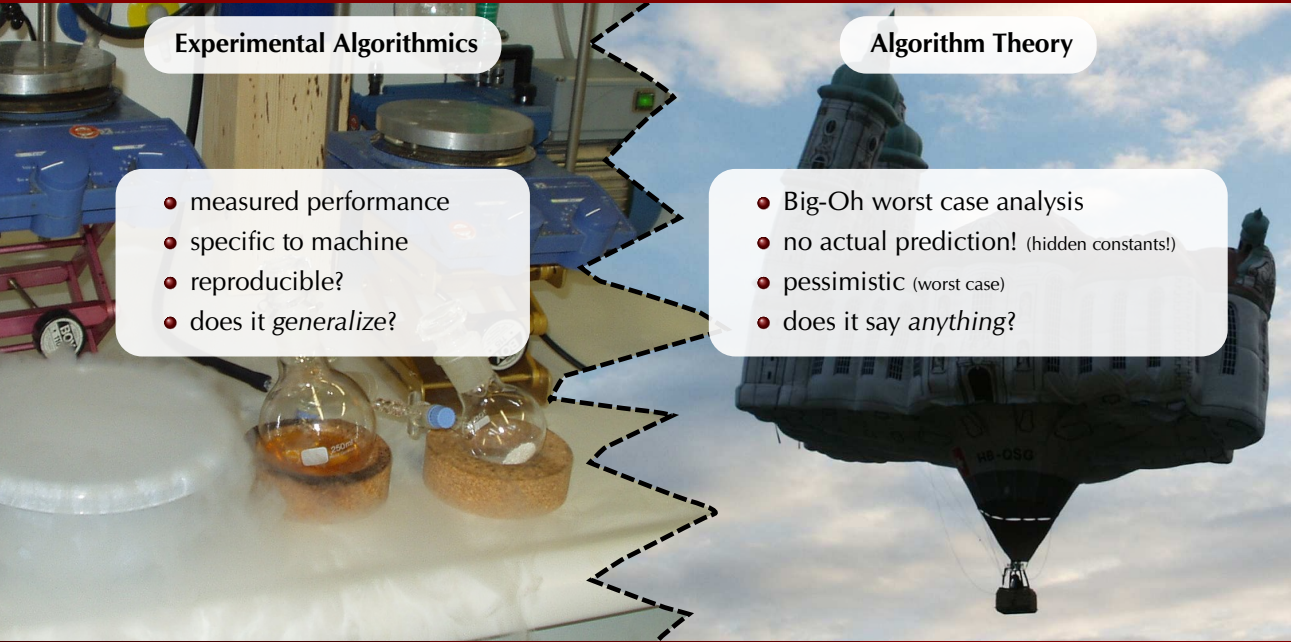
Experimental Algorithmics

- measured performance
- specific to machine
- reproducible?
- does it *generalize*?

Algorithm Theory

- Big-Oh worst case analysis
- no actual prediction! (hidden constants!)
- pessimistic (worst case)

Algorithm Science



Experimental Algorithmics

- measured performance
- specific to machine
- reproducible?
- does it *generalize*?

Algorithm Theory

- Big-Oh worst case analysis
- no actual prediction! (hidden constants!)
- pessimistic (worst case)
- does it say *anything*?

Algorithm Science

Experimental Algorithmics

- measured performance
- specific to machine
- reproducible?
- does it *generalize*?

Algorithm Theory

- Big-Oh worst case analysis
- no actual prediction! (hidden constants!)
- pessimistic (worst case)
- does it say *anything*?



Platform-independent performance guarantees

- concrete, useful predictions
- concrete, analyzable model
- validated by experiments

Case Study: Sorting in Java Library

Why Java?

- Open-Source reference implementation *OpenJDK*
- Basis for Oracle JRE, Android JRE $\rightsquigarrow$ might contain executed algorithms in existence!
- **within new millenium: all-new sorting methods!**

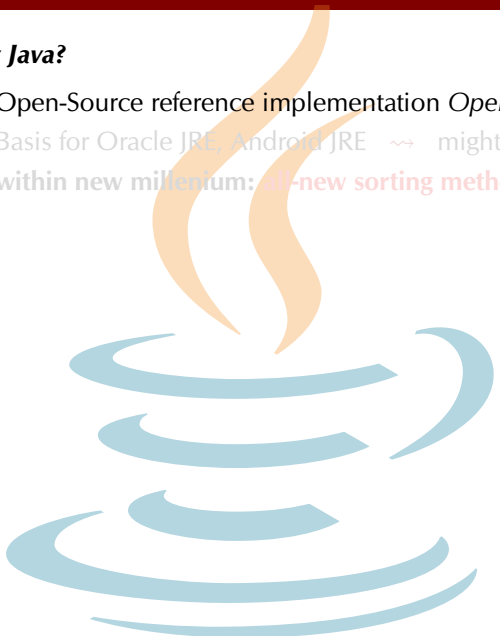


Java™

Case Study: Sorting in Java Library

Why Java?

- Open-Source reference implementation *OpenJDK*
- Basis for Oracle JRE, Android JRE $\rightsquigarrow$ might contain executed algorithms in existence!
- **within new millenium: all-new sorting methods!**

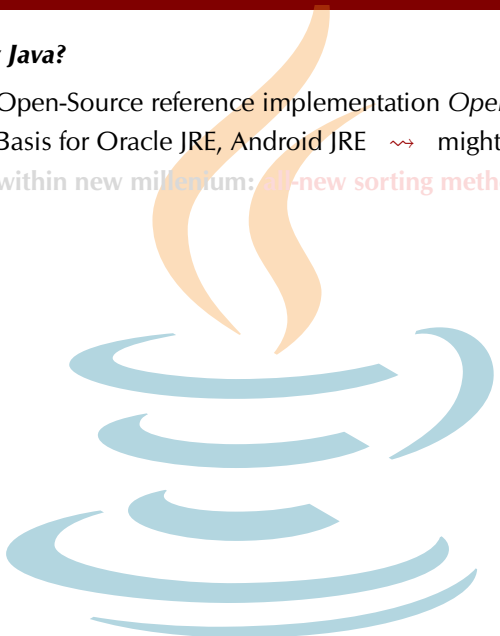


Java™

Case Study: Sorting in Java Library

Why Java?

- Open-Source reference implementation *OpenJDK*
- Basis for Oracle JRE, Android JRE $\rightsquigarrow$ might contain executed algorithms in existence!
- within new millenium: all-new sorting methods!

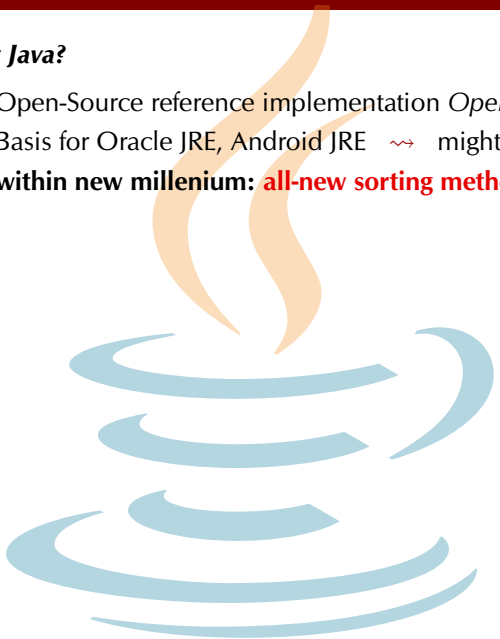


Java™

Case Study: Sorting in Java Library

Why Java?

- Open-Source reference implementation *OpenJDK*
- Basis for Oracle JRE, Android JRE $\rightsquigarrow$ might contain executed algorithms in existence!
- **within new millenium: all-new sorting methods!**



Java™

Case Study: Sorting in Java Library

Why Java?

- Open-Source reference implementation *OpenJDK*
- Basis for Oracle JRE, Android JRE $\rightsquigarrow$ might contain executed algorithms in existence!
- **within new millenium: all-new sorting methods!**

1 ***Dual-Pivot Quicksort***

- for primitive types (`int[]`, `double[]` etc.)
- 2008 Started as hobby project by *Vladimir Yaroslavskiy* (Sun Microsystems)
- since 2009 in OpenJDK



Case Study: Sorting in Java Library

Why Java?

- Open-Source reference implementation *OpenJDK*
- Basis for Oracle JRE, Android JRE $\rightsquigarrow$ might contain executed algorithms in existence!
- **within new millenium: all-new sorting methods!**

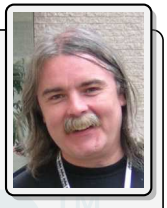
1 ***Dual-Pivot Quicksort***

- for primitive types (`int[]`, `double[]` etc.)
- 2008 Started as hobby project by *Vladimir Yaroslavskiy* (Sun Microsystems)
- since 2009 in OpenJDK



2 ***Timsort***

- for objects (`Object[]`)
- adaptive stable mergesort
- 2002 developed for `list.sort` in CPython by *Tim Peters*
- since 2009 in OpenJDK



Case Study: Sorting in Java Library

Why Java?

- Open-Source reference implementation *OpenJDK*
- Basis for Oracle JRE, Android JRE $\rightsquigarrow$ might contain executed algorithms in existence!
- **within new millenium: all-new sorting methods!**

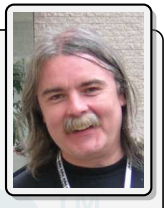
1 *Dual-Pivot Quicksort*

- for primitive types (`int[]`, `double[]` etc.)
- 2008 Started as hobby project by *Vladimir Yaroslavskiy* (Sun Microsystems)
- since 2009 in OpenJDK



2 *Timsort*

- for objects (`Object[]`)
- adaptive stable mergesort
- 2002 developed for `list.sort` in CPython by *Tim Peters*
- since 2009 in OpenJDK



- both algorithms from **developers**

Case Study: Sorting in Java Library

Why Java?

- Open-Source reference implementation *OpenJDK*
- Basis for Oracle JRE, Android JRE $\rightsquigarrow$ might contain executed algorithms in existence!
- **within new millenium: all-new sorting methods!**

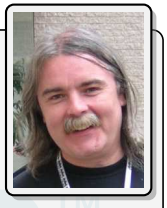
1 *Dual-Pivot Quicksort*

- for primitive types (`int[]`, `double[]` etc.)
- 2008 Started as hobby project by *Vladimir Yaroslavskiy* (Sun Microsystems)
- since 2009 in OpenJDK



2 *Timsort*

- for objects (`Object[]`)
- adaptive stable mergesort
- 2002 developed for `list.sort` in CPython by *Tim Peters*
- since 2009 in OpenJDK



- both algorithms from **developers**

$\rightsquigarrow$ **Focus:** Benchmarks & Unit-Tests not on proofs or analysis

Why a new Quicksort?

What happened in 2009? *Why retire a well-tested sorting method?*

- till 2009, Java used ***standard quicksort***
- Java 7 uses **new** ***dual-pivot quicksort***

Why a new Quicksort?

What happened in 2009? *Why retire a well-tested sorting method?*

- till 2009, Java used ***standard quicksort***
- Java 7 uses **new** ***dual-pivot quicksort***

Why a new Quicksort?

What happened in 2009? *Why retire a well-tested sorting method?*

- till 2009, Java used ***standard quicksort***
- Java 7 uses **new** ***dual-pivot quicksort***

Why a new Quicksort?

What happened in 2009? *Why retire a well-tested sorting method?*

crossing-pointer partitioning
+ median-of-three
+ Insertionsort

- till 2009, Java used ***standard quicksort***
- Java 7 uses **new** ***dual-pivot quicksort***

Why a new Quicksort?

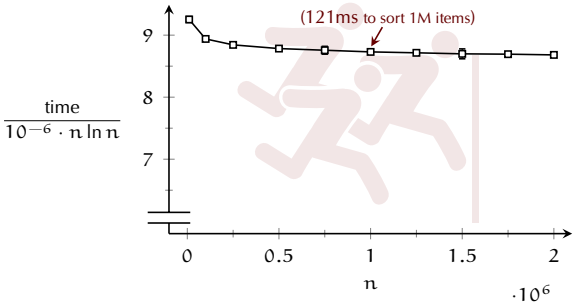
What happened in 2009? *Why retire a well-tested sorting method?*

Java 6 sorting method

crossing-pointer partitioning
+ median-of-three
+ Insertionsort

like (almost) all libraries

- till 2009, Java used ***standard quicksort***
- Java 7 uses **new** ***dual-pivot quicksort***



Normalized Java runtimes (in *ms*).

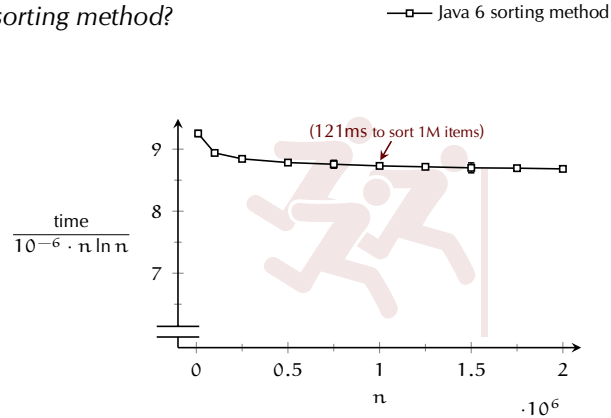
Average and standard deviation of
1000 random permutations per size.

Why a new Quicksort?

What happened in 2009? *Why retire a well-tested sorting method?*



- till 2009, Java used ***standard quicksort***
- Java 7 uses **new** ***dual-pivot quicksort***



Normalized Java runtimes (in *ms*).

Average and standard deviation of
1000 random permutations per size.

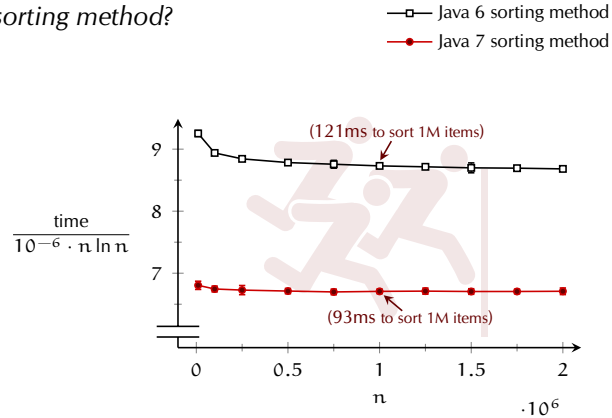
Why a new Quicksort?

What happened in 2009? *Why retire a well-tested sorting method?*

crossing-pointer partitioning
+ median-of-three
+ Insertionsort

like (almost) all libraries

- till 2009, Java used ***standard quicksort***
- Java 7 uses **new** ***dual-pivot quicksort***
  - 20 % faster than highly-tuned Java 6!



Normalized Java runtimes (in *ms*).

Average and standard deviation of
1000 random permutations per size.

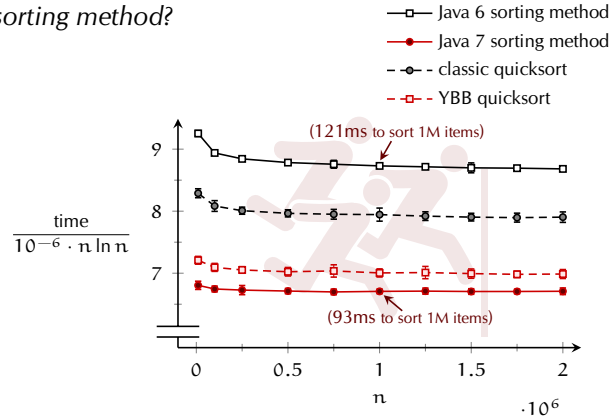
Why a new Quicksort?

What happened in 2009? *Why retire a well-tested sorting method?*

crossing-pointer partitioning
+ median-of-three
+ Insertionsort

like (almost) all libraries

- till 2009, Java used ***standard quicksort***
- Java 7 uses **new** ***dual-pivot quicksort***
  - 20 % faster than highly-tuned Java 6!
  - similarly for core algorithms!



Normalized Java runtimes (in *ms*).

Average and standard deviation of
1000 random permutations per size.

Why a new Quicksort?

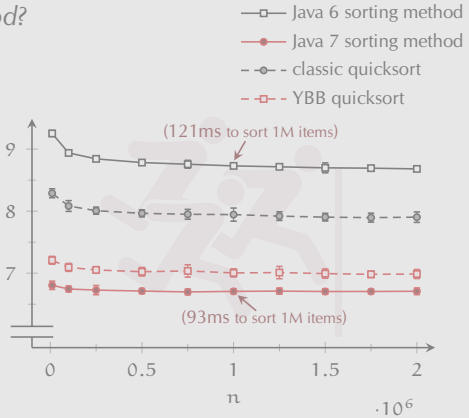
What happened in 2009? *Why retire a well-tested sorting method?*

crossing-pointer partitioning
+ median-of-three
+ Insertionsort

like (almost)
all libraries

- till 2009, Java used ***standard quicksort***
- Java 7 uses **new** ***dual-pivot quicksort***
  - 20 % faster than highly-tuned Java 6!
  - similarly for core algorithms!

No sound explanation for running time known in 2009!



Normalized Java runtimes (in *ms*).

Average and standard deviation of
1000 random permutations per size.

Why a new Quicksort?

What happened in 2009? *Why retire a well-tested sorting method?*

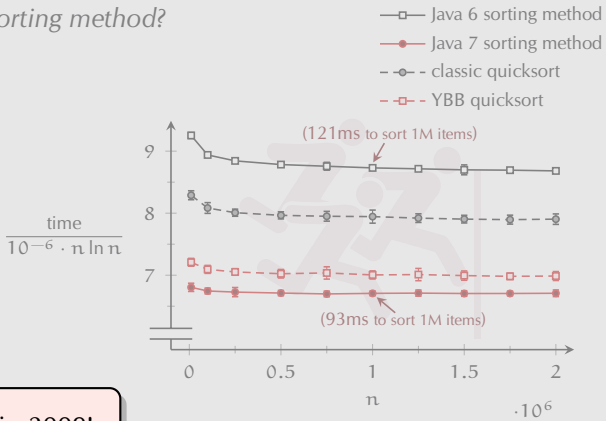
crossing-pointer partitioning
+ median-of-three
+ Insertionsort

like (almost)
all libraries

- till 2009, Java used ***standard quicksort***
- Java 7 uses **new** ***dual-pivot quicksort***
  - 20 % faster than highly-tuned Java 6!
  - similarly for core algorithms!

No sound explanation for running time known in 2009!

Is it an artifact of Java?



Normalized Java runtimes (in *ms*).

Average and standard deviation of
1000 random permutations per size.

Why a new Quicksort?

What happened in 2009? *Why retire a well-tested sorting method?*

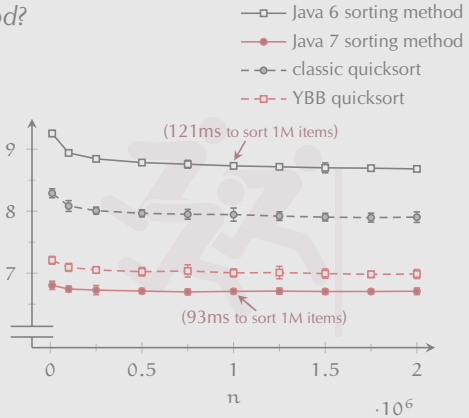
crossing-pointer partitioning
+ median-of-three
+ Insertionsort

like (almost)
all libraries

- till 2009, Java used ***standard quicksort***
- Java 7 uses **new** ***dual-pivot quicksort***
  - 20 % faster than highly-tuned Java 6!
  - similarly for core algorithms!

No sound explanation for running time known in 2009!

Is it an artifact of Java? Rather no; similar results in C++ …



Normalized Java runtimes (in *ms*).

Average and standard deviation of
1000 random permutations per size.

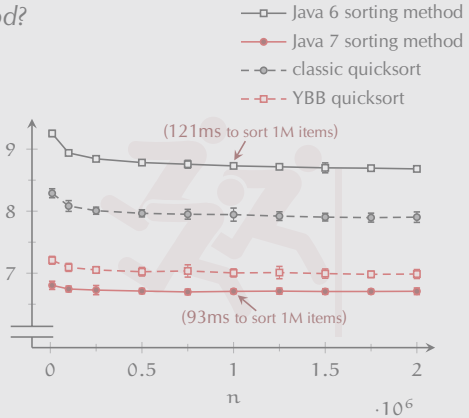
Why a new Quicksort?

What happened in 2009? *Why retire a well-tested sorting method?*

crossing-pointer partitioning
+ median-of-three
+ Insertionsort

like (almost)
all libraries

- till 2009, Java used ***standard quicksort***
- Java 7 uses **new** ***dual-pivot quicksort***
  - 20 % faster than highly-tuned Java 6!
  - similarly for core algorithms!



Normalized Java runtimes (in *ms*).

Average and standard deviation of
1000 random permutations per size.

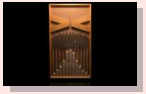
No sound explanation for running time known in 2009!

Is it an artifact of Java? Rather no; similar results in C++ …

Why did no-one come up with this earlier?

Outline







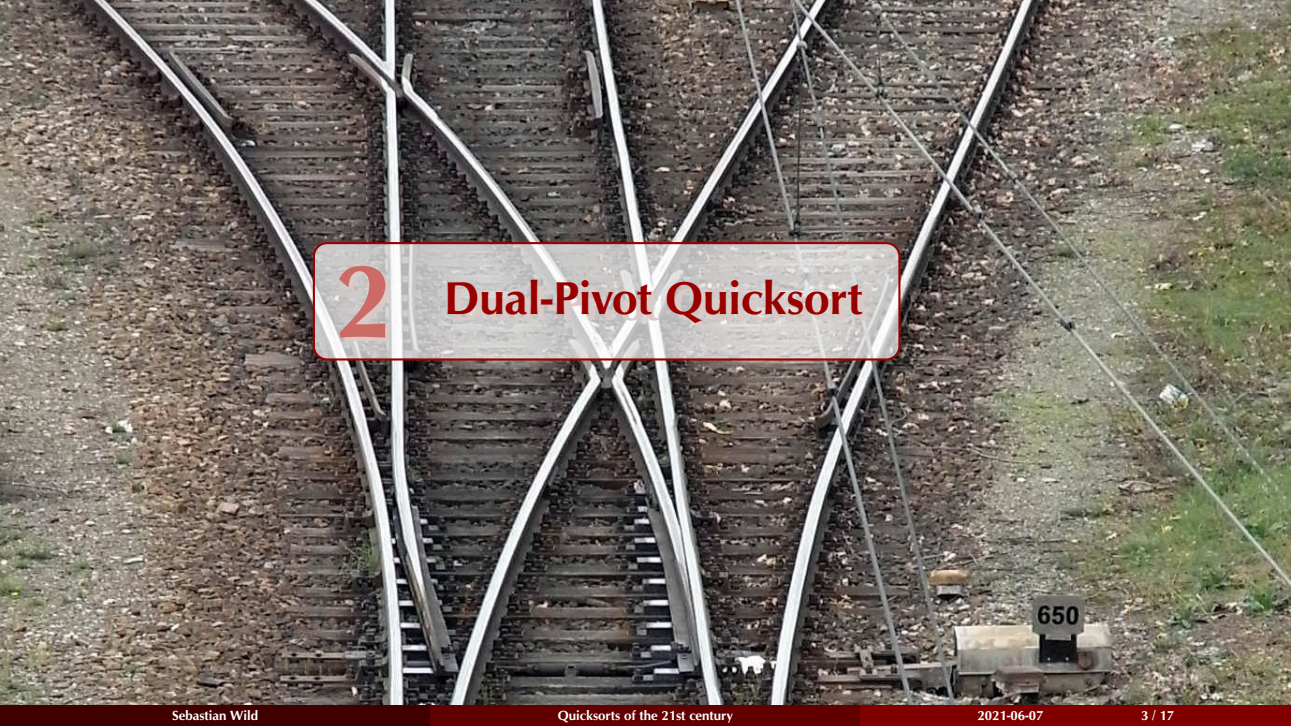
1 Context

2 Dual-Pivot Quicksort

3 Average-Case Analysis

4 Quicksort + X

5 Conclusion



2 Dual-Pivot Quicksort

Dual-Pivot Quicksort: Conceptual View

Standard Quicksort

1. Choose a pivot element P.
2. For each element x, determine its **class** by comparing x to P.
   - **small** if $x < P$
   - **large** if $P < x$
3. Rearrange elements according to classes:

4. Recurse on left & right subarray.

Dual-Pivot Quicksort: Conceptual View

Standard Quicksort

1. Choose a pivot element P.
2. For each element x, determine its **class** by comparing x to P.
   - **small** if $x < P$
   - **large** if $P < x$
3. Rearrange elements according to classes:

4. Recurse on left & right subarray.

Dual-Pivot Quicksort: Conceptual View

Standard Quicksort

1. Choose a pivot element P.
2. For each element x, determine its **class** by comparing x to P.
   - **small** if $x < P$
   - **large** if $P < x$
3. Rearrange elements according to classes:

4. Recurse on left & right subarray.

Dual-Pivot Quicksort: Conceptual View

Standard Quicksort

1. Choose a pivot element P.
2. For each element x, determine its **class** by comparing x to P.
   - **small** if $x < P$
   - **large** if $P < x$

   assuming distinct elements in this talk
3. Rearrange elements according to classes:

4. Recurse on left & right subarray.

Dual-Pivot Quicksort: Conceptual View

Standard Quicksort

1. Choose a pivot element P.
2. For each element x, determine its **class** by comparing x to P.
   - **small** if $x < P$
   - **large** if $P < x$

   assuming distinct elements in this talk
3. Rearrange elements according to classes:



4. Recurse on left & right subarray.

Dual-Pivot Quicksort: Conceptual View

Standard Quicksort

1. Choose a pivot element P.
2. For each element x, determine its **class** by comparing x to P.
   - **small** if $x < P$
   - **large** if $P < x$

   assuming distinct elements in this talk
3. Rearrange elements according to classes:



4. Recurse on left & right subarray.

Dual-Pivot Quicksort: Conceptual View

Standard Quicksort

1. Choose a pivot element P.
2. For each element x, determine its **class** by comparing x to P.
   - **small** if $x < P$
   - **large** if $P < x$

   assuming distinct elements in this talk
3. Rearrange elements according to classes:



4. Recurse on left & right subarray.

Dual-Pivot Quicksort

1. Choose **two pivots** $P_1 \leqslant P_2$
2. For each element x, determine its **class** by comparing x to **pivots** P_1 and P_2
   - **small** for $x < P_1$
   - **medium** for $P_1 < x < P_2$
   - **large** for $P_2 < x$
3. Arrange elements according to classes:

4. Sort subarrays recursively.

Dual-Pivot Quicksort: Conceptual View

Standard Quicksort

1. Choose a pivot element P.
2. For each element x, determine its **class** by comparing x to P.
   - **small** if $x < P$
   - **large** if $P < x$

   assuming distinct elements in this talk
3. Rearrange elements according to classes:



4. Recurse on left & right subarray.

Dual-Pivot Quicksort

1. Choose **two pivots** $P_1 \leqslant P_2$
2. For each element x, determine its **class** by comparing x to **pivots** P_1 and P_2
   - **small** for $x < P_1$
   - **medium** for $P_1 < x < P_2$
   - **large** for $P_2 < x$
3. Arrange elements according to classes:

4. Sort subarrays recursively.

Dual-Pivot Quicksort: Conceptual View

Standard Quicksort

1. Choose a pivot element P.
2. For each element x, determine its **class** by comparing x to P.
   - **small** if $x < P$
   - **large** if $P < x$

   assuming distinct elements in this talk
3. Rearrange elements according to classes:



4. Recurse on left & right subarray.

Dual-Pivot Quicksort

1. Choose **two pivots** $P_1 \leqslant P_2$
2. For each element x, determine its **class** by comparing x to **pivots** P_1 and P_2
   - **small** for $x < P_1$
   - **medium** for $P_1 < x < P_2$
   - **large** for $P_2 < x$
3. Arrange elements according to classes:

4. Sort subarrays recursively.

Dual-Pivot Quicksort: Conceptual View

Standard Quicksort

1. Choose a pivot element P.
2. For each element x, determine its **class** by comparing x to P.
   - **small** if $x < P$
   - **large** if $P < x$

   assuming distinct elements in this talk
3. Rearrange elements according to classes:



4. Recurse on left & right subarray.

Dijkstra's Dutch National Flag Problem

Dual-Pivot Quicksort

1. Choose **two pivots** $P_1 \leqslant P_2$
2. For each element x, determine its **class** by comparing x to **pivots** P_1 and P_2
   - **small** for $x < P_1$
   - **medium** for $P_1 < x < P_2$
   - **large** for $P_2 < x$
3. Arrange elements according to classes:



4. Sort subarrays recursively.

Dual-Pivot Quicksort: Conceptual View

Standard Quicksort

1. Choose a pivot element P.
2. For each element x, determine its **class** by comparing x to P.
   - **small** if $x < P$
   - **large** if $P < x$

   assuming distinct elements in this talk
3. Rearrange elements according to classes:



4. Recurse on left & right subarray.

Dijkstra's Dutch National Flag Problem

Dual-Pivot Quicksort

1. Choose **two pivots** $P_1 \leqslant P_2$
2. For each element x, determine its **class** by comparing x to **pivots** P_1 and P_2
   - **small** for $x < P_1$
   - **medium** for $P_1 < x < P_2$
   - **large** for $P_2 < x$
3. Arrange elements according to classes:



4. Sort subarrays recursively.

Dual-Pivot Quicksort: Conceptual View

Standard Quicksort

1. Choose a pivot
2. For each element x, determine its **class** by comparing x to P.
   - **small** if $x < P$
   - **large** if $P < x$

assuming distinct elements in this talk

3. Rearrange elements according to classes:



4. Recurse on left & right subarray.

How to do ❸ efficiently and in place? ⇝ *YBB Partitioning*

Dual-Pivot Quicksort

2. For each element x, determine its **class** by comparing x to **pivots** P_1 and P_2
   - **small** for $x < P_1$
   - **medium** for $P_1 < x < P_2$
   - **large** for $P_2 < x$

Dijkstra's Dutch National Flag Problem

3. Arrange elements according to classes:



4. Sort subarrays recursively.

Dual-Pivot Quicksort: Conceptual View

Standard Quicksort

1. Choose a pivot
2. For each element x, determine its **class** by comparing x to P.
   - **small** if $x < P$
   - **large** if $P < x$

   assuming distinct elements in this talk
3. Rearrange elements according to classes:



4. Recurse on left & right subarray.

How to do ❸ efficiently and in place? ⇝ ***YBB Partitioning***

Dual-Pivot Quicksort

2. For each element x, determine its **class** by comparing x to **pivots** P_1 and P_2
   - **small** for $x < P_1$
   - **medium** for $P_1 < x < P_2$
   - **large** for $P_2 < x$

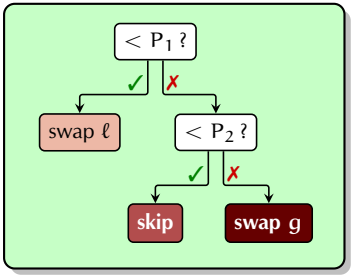
Dijkstra's Dutch National Flag Problem

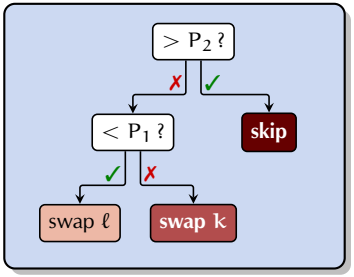
3. Arrange elements according to classes:



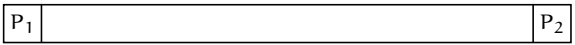
4. Sort subarrays recursively.

YBB Partitioning Example



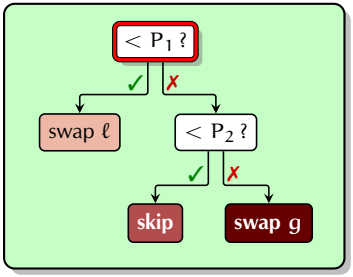


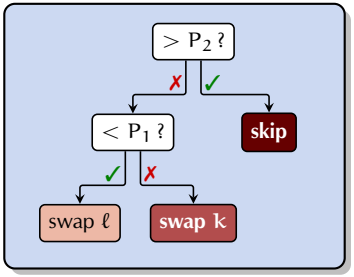


Invariant: 

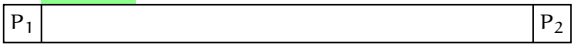

```
YBBQUICKSORT(A, left, right)
1   if right ⩽ left then return
2   P1 := A[left];    P2 := A[right]
3   if P1 > P2 then Swap P1 and P2 end if
4   ℓ := left + 1;    g := right − 1;    k := ℓ
5   while k ⩽ g
6       if A[k] < P1
7           Swap A[k] and A[ℓ] ;  ℓ := ℓ + 1
8       else if A[k] ⩾ P2
9           while A[g] > P2 and k < g
10              g := g − 1
11          end while
12          Swap A[k] and A[g] ;  g := g − 1
13          if A[k] < P1
14              Swap A[k] and A[ℓ] ;  ℓ := ℓ + 1
15          end if
16      end if
17      k := k + 1
18  end while
19  ℓ := ℓ − 1;    g := g + 1
20  Swap A[left] and A[ℓ] ; Swap A[right] and A[g]
21  YBBQUICKSORT(A, left , ℓ − 1)
22  YBBQUICKSORT(A, ℓ + 1, g − 1)
23  YBBQUICKSORT(A, g + 1, right )
```

YBB Partitioning Example





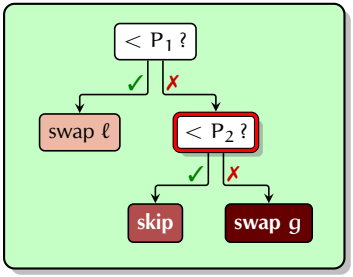


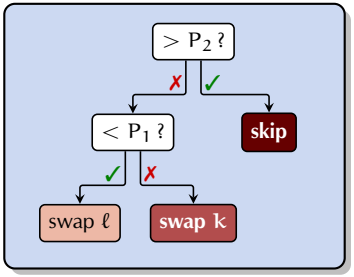
Invariant: 

YBBQUICKSORT(A, *left*, *right*)

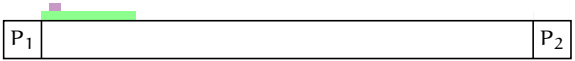
```
1  if right ⩽ left then return
2  P1 := A[left];    P2 := A[right]
3  if P1 > P2 then Swap P1 and P2 end if
4  ℓ := left + 1;    g := right − 1;    k := ℓ
5  while k ⩽ g
6      if A[k] < P1
7          Swap A[k] and A[ℓ] ;  ℓ := ℓ + 1
8      else if A[k] ⩾ P2
9          while A[g] > P2 and k < g
10             g := g − 1
11         end while
12         Swap A[k] and A[g] ;  g := g − 1
13         if A[k] < P1
14             Swap A[k] and A[ℓ] ;  ℓ := ℓ + 1
15         end if
16     end if
17     k := k + 1
18 end while
19 ℓ := ℓ − 1;    g := g + 1
20 Swap A[left] and A[ℓ] ; Swap A[right] and A[g]
21 YBBQUICKSORT(A, left , ℓ − 1)
22 YBBQUICKSORT(A, ℓ + 1, g − 1)
23 YBBQUICKSORT(A, g + 1, right )
```

YBB Partitioning Example



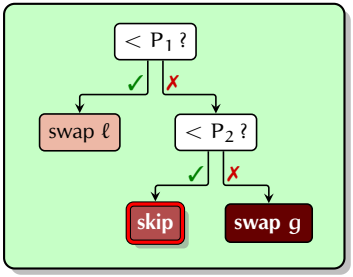


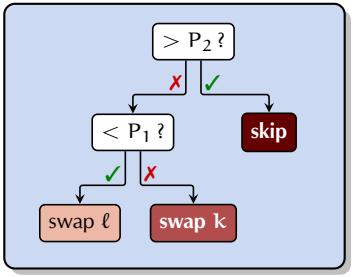


Invariant: 


```
YBBQUICKSORT(A, left, right)
1  if right ⩽ left then return
2  P1 := A[left];    P2 := A[right]
3  if P1 > P2 then Swap P1 and P2 end if
4  ℓ := left + 1;    g := right − 1;    k := ℓ
5  while k ⩽ g
6      if A[k] < P1
7          Swap A[k] and A[ℓ] ;  ℓ := ℓ + 1
8      else if A[k] ⩾ P2
9          while A[g] > P2 and k < g
10             g := g − 1
11         end while
12         Swap A[k] and A[g] ;  g := g − 1
13         if A[k] < P1
14             Swap A[k] and A[ℓ] ;  ℓ := ℓ + 1
15         end if
16     end if
17     k := k + 1
18 end while
19 ℓ := ℓ − 1;    g := g + 1
20 Swap A[left] and A[ℓ] ; Swap A[right] and A[g]
21 YBBQUICKSORT(A, left , ℓ − 1)
22 YBBQUICKSORT(A, ℓ + 1, g − 1)
23 YBBQUICKSORT(A, g + 1, right )
```

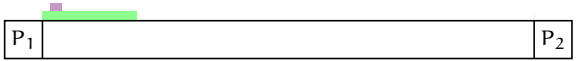
YBB Partitioning Example







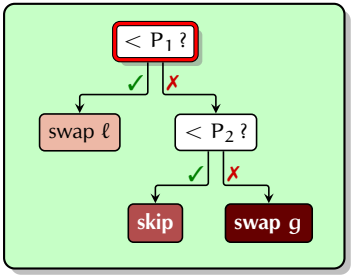
Invariant:

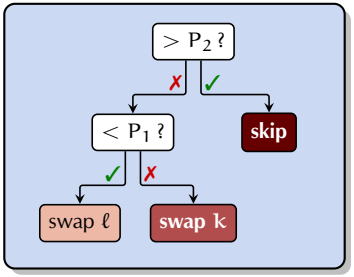


YBBQUICKSORT(A, *left*, *right*)

```
1  if right ⩽ left then return
2  P1 := A[left];    P2 := A[right]
3  if P1 > P2 then Swap P1 and P2 end if
4  ℓ := left + 1;    g := right − 1;    k := ℓ
5  while k ⩽ g
6      if A[k] < P1
7          Swap A[k] and A[ℓ] ;  ℓ := ℓ + 1
8      else if A[k] ⩾ P2
9          while A[g] > P2 and k < g
10             g := g − 1
11         end while
12         Swap A[k] and A[g] ;  g := g − 1
13         if A[k] < P1
14             Swap A[k] and A[ℓ] ;  ℓ := ℓ + 1
15         end if
16     end if
17     k := k + 1
18 end while
19 ℓ := ℓ − 1;    g := g + 1
20 Swap A[left] and A[ℓ] ; Swap A[right] and A[g]
21 YBBQUICKSORT(A, left , ℓ − 1)
22 YBBQUICKSORT(A, ℓ + 1, g − 1)
23 YBBQUICKSORT(A, g + 1, right )
```

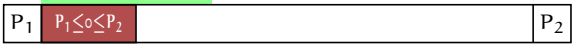
YBB Partitioning Example







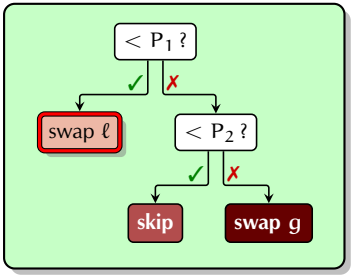
Invariant:

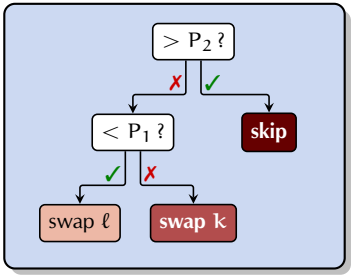


YBBQUICKSORT(A, *left*, *right*)

```
1  if right ⩽ left then return
2  P1 := A[left];    P2 := A[right]
3  if P1 > P2 then Swap P1 and P2 end if
4  ℓ := left + 1;    g := right − 1;    k := ℓ
5  while k ⩽ g
6      if A[k] < P1
7          Swap A[k] and A[ℓ] ;  ℓ := ℓ + 1
8      else if A[k] ⩾ P2
9          while A[g] > P2 and k < g
10             g := g − 1
11         end while
12         Swap A[k] and A[g] ;  g := g − 1
13         if A[k] < P1
14             Swap A[k] and A[ℓ] ;  ℓ := ℓ + 1
15         end if
16     end if
17     k := k + 1
18 end while
19 ℓ := ℓ − 1;    g := g + 1
20 Swap A[left] and A[ℓ] ; Swap A[right] and A[g]
21 YBBQUICKSORT(A, left , ℓ − 1)
22 YBBQUICKSORT(A, ℓ + 1, g − 1)
23 YBBQUICKSORT(A, g + 1, right )
```

YBB Partitioning Example



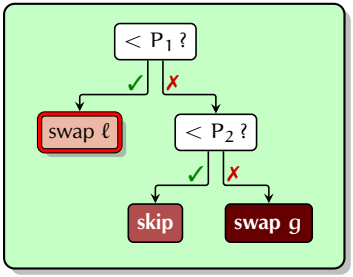


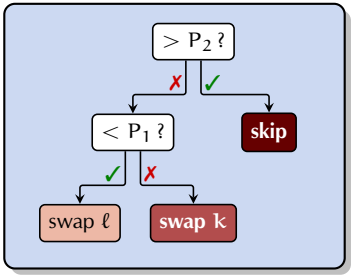


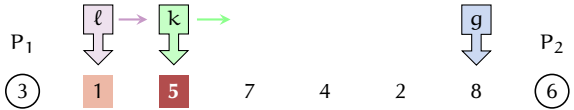
Invariant: $P_1 \le o \le P_2$

```
YBBQUICKSORT(A, left, right)
1  if right ⩽ left then return
2  P1 := A[left];    P2 := A[right]
3  if P1 > P2 then Swap P1 and P2 end if
4  ℓ := left + 1;    g := right − 1;    k := ℓ
5  while k ⩽ g
6      if A[k] < P1
7          Swap A[k] and A[ℓ] ;  ℓ := ℓ + 1
8      else if A[k] ⩾ P2
9          while A[g] > P2 and k < g
10             g := g − 1
11         end while
12         Swap A[k] and A[g] ;  g := g − 1
13         if A[k] < P1
14             Swap A[k] and A[ℓ] ;  ℓ := ℓ + 1
15         end if
16     end if
17     k := k + 1
18 end while
19 ℓ := ℓ − 1;    g := g + 1
20 Swap A[left] and A[ℓ] ; Swap A[right] and A[g]
21 YBBQUICKSORT(A, left , ℓ − 1)
22 YBBQUICKSORT(A, ℓ + 1, g − 1)
23 YBBQUICKSORT(A, g + 1, right )
```

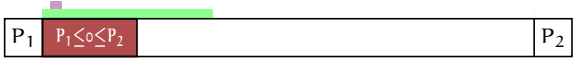
YBB Partitioning Example







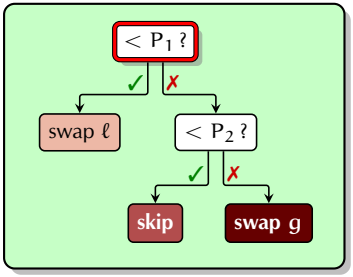
Invariant:

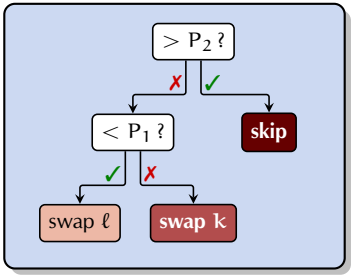


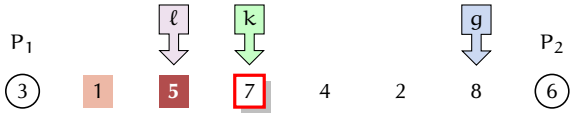
YBBQUICKSORT(A, *left*, *right*)

```
1  if right ⩽ left then return
2  P1 := A[left];    P2 := A[right]
3  if P1 > P2 then Swap P1 and P2 end if
4  ℓ := left + 1;    g := right − 1;    k := ℓ
5  while k ⩽ g
6      if A[k] < P1
7          Swap A[k] and A[ℓ] ;  ℓ := ℓ + 1
8      else if A[k] ⩾ P2
9          while A[g] > P2 and k < g
10             g := g − 1
11         end while
12         Swap A[k] and A[g] ;  g := g − 1
13         if A[k] < P1
14             Swap A[k] and A[ℓ] ;  ℓ := ℓ + 1
15         end if
16     end if
17     k := k + 1
18 end while
19 ℓ := ℓ − 1;    g := g + 1
20 Swap A[left] and A[ℓ] ; Swap A[right] and A[g]
21 YBBQUICKSORT(A, left , ℓ − 1)
22 YBBQUICKSORT(A, ℓ + 1, g − 1)
23 YBBQUICKSORT(A, g + 1, right )
```

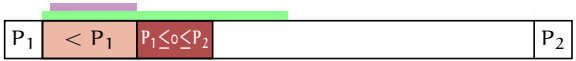
YBB Partitioning Example





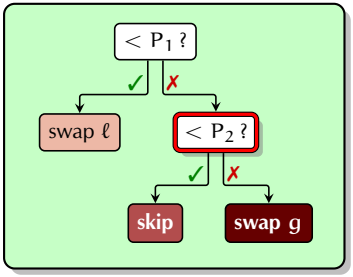


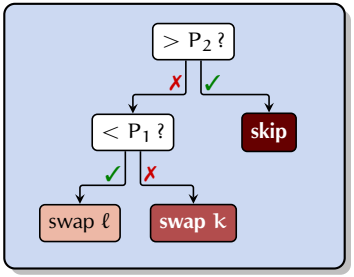
Invariant:

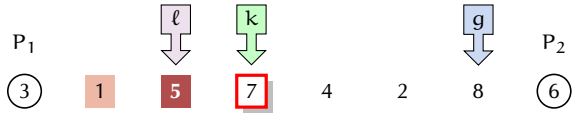


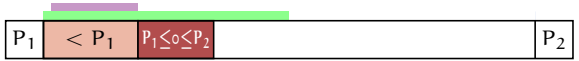

```
YBBQUICKSORT(A, left, right)
1  if right ⩽ left then return
2  P1 := A[left];    P2 := A[right]
3  if P1 > P2 then Swap P1 and P2 end if
4  ℓ := left + 1;    g := right − 1;    k := ℓ
5  while k ⩽ g
6      if A[k] < P1
7          Swap A[k] and A[ℓ] ;  ℓ := ℓ + 1
8      else if A[k] ⩾ P2
9          while A[g] > P2 and k < g
10             g := g − 1
11         end while
12         Swap A[k] and A[g] ;  g := g − 1
13         if A[k] < P1
14             Swap A[k] and A[ℓ] ;  ℓ := ℓ + 1
15         end if
16     end if
17     k := k + 1
18 end while
19 ℓ := ℓ − 1;    g := g + 1
20 Swap A[left] and A[ℓ] ; Swap A[right] and A[g]
21 YBBQUICKSORT(A, left , ℓ − 1)
22 YBBQUICKSORT(A, ℓ + 1, g − 1)
23 YBBQUICKSORT(A, g + 1, right )
```

YBB Partitioning Example





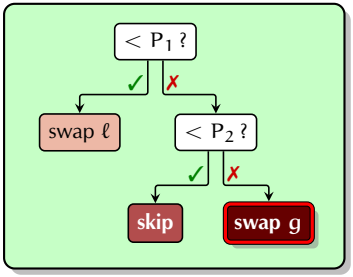


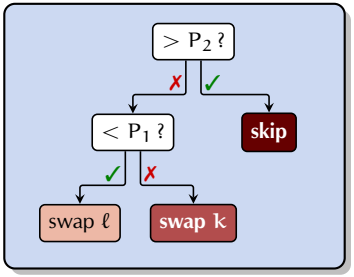


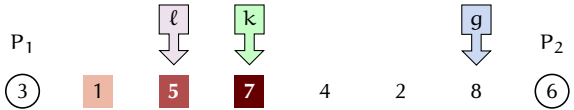
YBBQUICKSORT(A, *left*, *right*)

```
1  if right ⩽ left then return
2  P1 := A[left];    P2 := A[right]
3  if P1 > P2 then Swap P1 and P2 end if
4  ℓ := left + 1;    g := right − 1;    k := ℓ
5  while k ⩽ g
6      if A[k] < P1
7          Swap A[k] and A[ℓ] ;  ℓ := ℓ + 1
8      else if A[k] ⩾ P2
9          while A[g] > P2 and k < g
10             g := g − 1
11         end while
12         Swap A[k] and A[g] ;  g := g − 1
13         if A[k] < P1
14             Swap A[k] and A[ℓ] ;  ℓ := ℓ + 1
15         end if
16     end if
17     k := k + 1
18 end while
19 ℓ := ℓ − 1;    g := g + 1
20 Swap A[left] and A[ℓ] ; Swap A[right] and A[g]
21 YBBQUICKSORT(A, left , ℓ − 1)
22 YBBQUICKSORT(A, ℓ + 1, g − 1)
23 YBBQUICKSORT(A, g + 1, right )
```

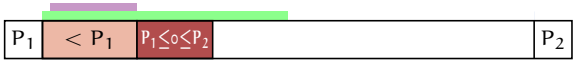
YBB Partitioning Example





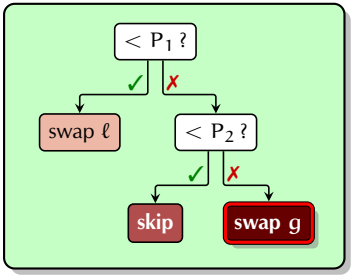


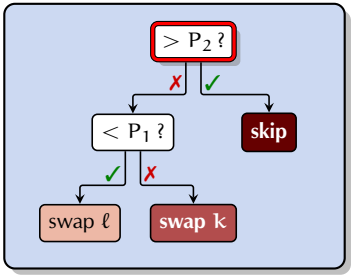
Invariant:

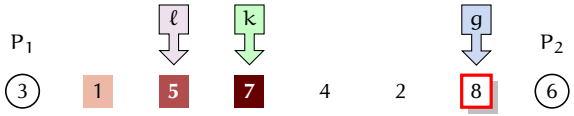



```
YBBQUICKSORT(A, left, right)
1  if right ⩽ left then return
2  P1 := A[left];    P2 := A[right]
3  if P1 > P2 then Swap P1 and P2 end if
4  ℓ := left + 1;    g := right − 1;    k := ℓ
5  while k ⩽ g
6      if A[k] < P1
7          Swap A[k] and A[ℓ] ;  ℓ := ℓ + 1
8      else if A[k] ⩾ P2
9          while A[g] > P2 and k < g
10             g := g − 1
11         end while
12         Swap A[k] and A[g] ;  g := g − 1
13         if A[k] < P1
14             Swap A[k] and A[ℓ] ;  ℓ := ℓ + 1
15         end if
16     end if
17     k := k + 1
18 end while
19 ℓ := ℓ − 1;    g := g + 1
20 Swap A[left] and A[ℓ] ; Swap A[right] and A[g]
21 YBBQUICKSORT(A, left , ℓ − 1)
22 YBBQUICKSORT(A, ℓ + 1, g − 1)
23 YBBQUICKSORT(A, g + 1, right )
```

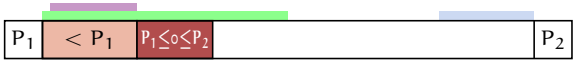
YBB Partitioning Example





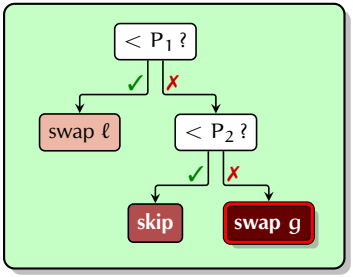


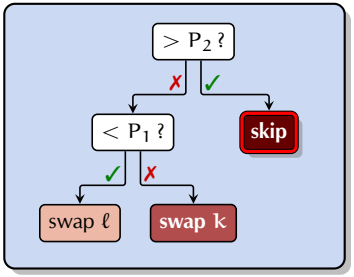
Invariant:

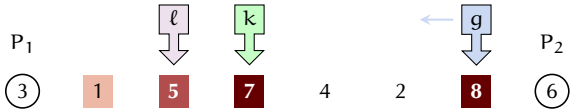



```
YBBQUICKSORT(A, left, right)
1  if right ⩽ left then return
2  P1 := A[left];    P2 := A[right]
3  if P1 > P2 then Swap P1 and P2 end if
4  ℓ := left + 1;    g := right − 1;    k := ℓ
5  while k ⩽ g
6      if A[k] < P1
7          Swap A[k] and A[ℓ] ;  ℓ := ℓ + 1
8      else if A[k] ⩾ P2
9          while A[g] > P2 and k < g
10             g := g − 1
11         end while
12         Swap A[k] and A[g] ;  g := g − 1
13         if A[k] < P1
14             Swap A[k] and A[ℓ] ;  ℓ := ℓ + 1
15         end if
16     end if
17     k := k + 1
18 end while
19 ℓ := ℓ − 1;    g := g + 1
20 Swap A[left] and A[ℓ] ; Swap A[right] and A[g]
21 YBBQUICKSORT(A, left , ℓ − 1)
22 YBBQUICKSORT(A, ℓ + 1, g − 1)
23 YBBQUICKSORT(A, g + 1, right )
```

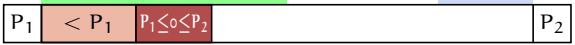
YBB Partitioning Example







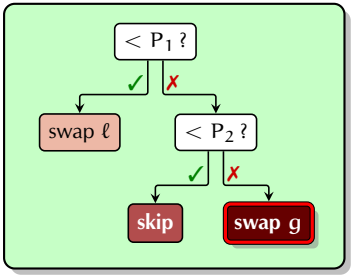
Invariant:

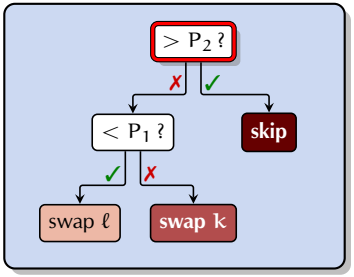


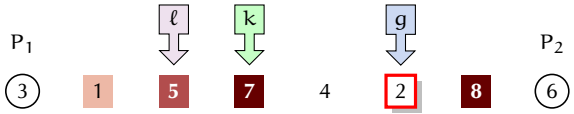
YBBQUICKSORT(A, *left*, *right*)

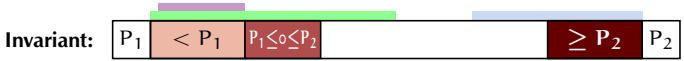
```
1  if right ⩽ left then return
2  P1 := A[left];    P2 := A[right]
3  if P1 > P2 then Swap P1 and P2 end if
4  ℓ := left + 1;    g := right − 1;    k := ℓ
5  while k ⩽ g
6      if A[k] < P1
7          Swap A[k] and A[ℓ] ;  ℓ := ℓ + 1
8      else if A[k] ⩾ P2
9          while A[g] > P2 and k < g
10             g := g − 1
11         end while
12         Swap A[k] and A[g] ;  g := g − 1
13         if A[k] < P1
14             Swap A[k] and A[ℓ] ;  ℓ := ℓ + 1
15         end if
16     end if
17     k := k + 1
18 end while
19 ℓ := ℓ − 1;    g := g + 1
20 Swap A[left] and A[ℓ] ; Swap A[right] and A[g]
21 YBBQUICKSORT(A, left , ℓ − 1)
22 YBBQUICKSORT(A, ℓ + 1, g − 1)
23 YBBQUICKSORT(A, g + 1, right )
```

YBB Partitioning Example



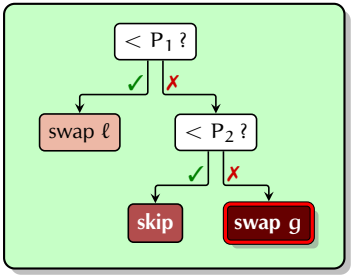


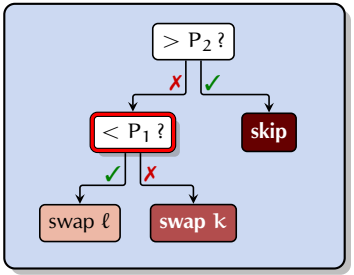


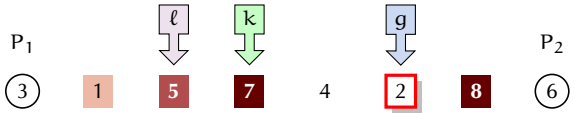



```
YBBQUICKSORT(A, left, right)
1  if right ⩽ left then return
2  P1 := A[left];    P2 := A[right]
3  if P1 > P2 then Swap P1 and P2 end if
4  ℓ := left + 1;    g := right − 1;    k := ℓ
5  while k ⩽ g
6      if A[k] < P1
7          Swap A[k] and A[ℓ] ;  ℓ := ℓ + 1
8      else if A[k] ⩾ P2
9          while A[g] > P2 and k < g
10             g := g − 1
11         end while
12         Swap A[k] and A[g] ;  g := g − 1
13         if A[k] < P1
14             Swap A[k] and A[ℓ] ;  ℓ := ℓ + 1
15         end if
16     end if
17     k := k + 1
18 end while
19 ℓ := ℓ − 1;    g := g + 1
20 Swap A[left] and A[ℓ] ; Swap A[right] and A[g]
21 YBBQUICKSORT(A, left , ℓ − 1)
22 YBBQUICKSORT(A, ℓ + 1, g − 1)
23 YBBQUICKSORT(A, g + 1, right )
```

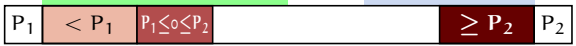
YBB Partitioning Example







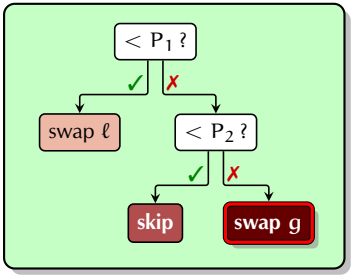
Invariant:

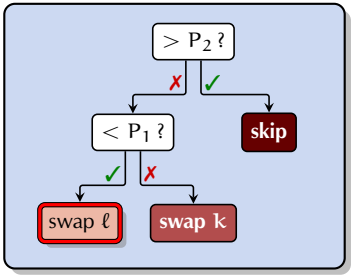


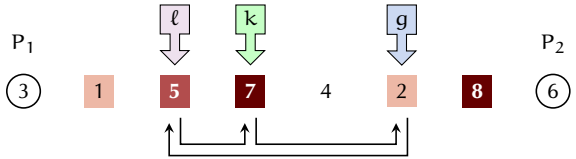
YBBQUICKSORT(A, *left*, *right*)

```
1  if right ⩽ left then return
2  P1 := A[left];    P2 := A[right]
3  if P1 > P2 then Swap P1 and P2 end if
4  ℓ := left + 1;    g := right − 1;    k := ℓ
5  while k ⩽ g
6      if A[k] < P1
7          Swap A[k] and A[ℓ] ;  ℓ := ℓ + 1
8      else if A[k] ⩾ P2
9          while A[g] > P2 and k < g
10             g := g − 1
11         end while
12         Swap A[k] and A[g] ;  g := g − 1
13     → if A[k] < P1
14             Swap A[k] and A[ℓ] ;  ℓ := ℓ + 1
15         end if
16     end if
17     k := k + 1
18 end while
19 ℓ := ℓ − 1;    g := g + 1
20 Swap A[left] and A[ℓ] ; Swap A[right] and A[g]
21 YBBQUICKSORT(A, left , ℓ − 1)
22 YBBQUICKSORT(A, ℓ + 1, g − 1)
23 YBBQUICKSORT(A, g + 1, right )
```

YBB Partitioning Example

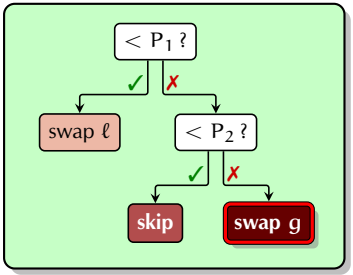


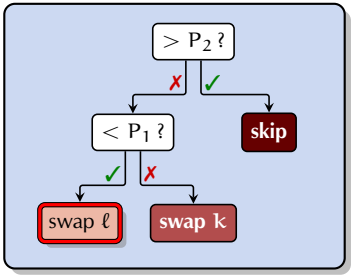


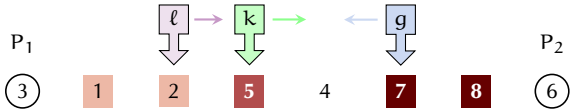


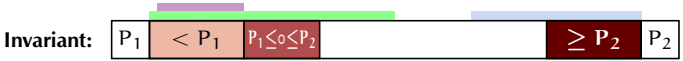

```
YBBQUICKSORT(A, left, right)
1  if right ⩽ left then return
2  P1 := A[left];    P2 := A[right]
3  if P1 > P2 then Swap P1 and P2 end if
4  ℓ := left + 1;    g := right − 1;    k := ℓ
5  while k ⩽ g
6      if A[k] < P1
7          Swap A[k] and A[ℓ] ;  ℓ := ℓ + 1
8      else if A[k] ⩾ P2
9          while A[g] > P2 and k < g
10             g := g − 1
11         end while
12         Swap A[k] and A[g] ;  g := g − 1
13         if A[k] < P1
14             Swap A[k] and A[ℓ] ;  ℓ := ℓ + 1
15         end if
16     end if
17     k := k + 1
18 end while
19 ℓ := ℓ − 1;    g := g + 1
20 Swap A[left] and A[ℓ] ; Swap A[right] and A[g]
21 YBBQUICKSORT(A, left , ℓ − 1)
22 YBBQUICKSORT(A, ℓ + 1, g − 1)
23 YBBQUICKSORT(A, g + 1, right )
```

YBB Partitioning Example



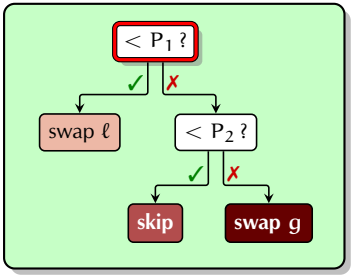


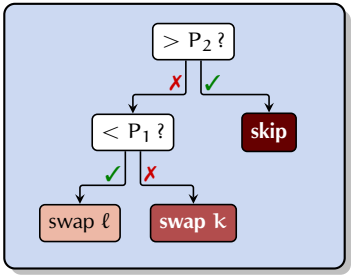


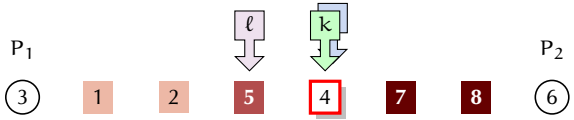


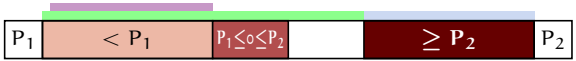

```
YBBQUICKSORT(A, left, right)
1  if right ⩽ left then return
2  P1 := A[left];    P2 := A[right]
3  if P1 > P2 then Swap P1 and P2 end if
4  ℓ := left + 1;    g := right − 1;    k := ℓ
5  while k ⩽ g
6      if A[k] < P1
7          Swap A[k] and A[ℓ] ;  ℓ := ℓ + 1
8      else if A[k] ⩾ P2
9          while A[g] > P2 and k < g
10             g := g − 1
11         end while
12         Swap A[k] and A[g] ;  g := g − 1
13         if A[k] < P1
14             Swap A[k] and A[ℓ] ;  ℓ := ℓ + 1
15         end if
16     end if
17     k := k + 1
18 end while
19 ℓ := ℓ − 1;    g := g + 1
20 Swap A[left] and A[ℓ] ; Swap A[right] and A[g]
21 YBBQUICKSORT(A, left , ℓ − 1)
22 YBBQUICKSORT(A, ℓ + 1, g − 1)
23 YBBQUICKSORT(A, g + 1, right )
```

YBB Partitioning Example



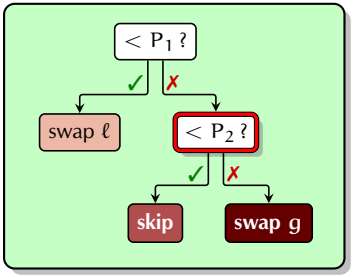


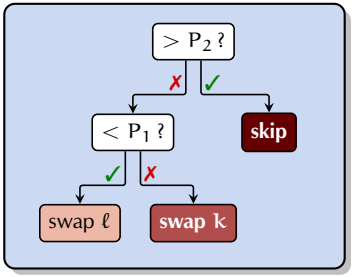


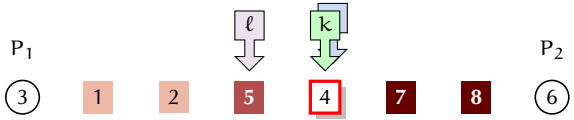


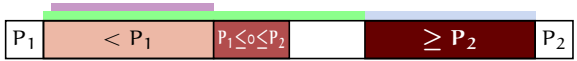

```
YBBQUICKSORT(A, left, right)
1  if right ⩽ left then return
2  P1 := A[left];    P2 := A[right]
3  if P1 > P2 then Swap P1 and P2 end if
4  ℓ := left + 1;    g := right − 1;    k := ℓ
5  while k ⩽ g
6      if A[k] < P1
7          Swap A[k] and A[ℓ] ;  ℓ := ℓ + 1
8      else if A[k] ⩾ P2
9          while A[g] > P2 and k < g
10             g := g − 1
11         end while
12         Swap A[k] and A[g] ;  g := g − 1
13         if A[k] < P1
14             Swap A[k] and A[ℓ] ;  ℓ := ℓ + 1
15         end if
16     end if
17     k := k + 1
18 end while
19 ℓ := ℓ − 1;    g := g + 1
20 Swap A[left] and A[ℓ] ; Swap A[right] and A[g]
21 YBBQUICKSORT(A, left , ℓ − 1)
22 YBBQUICKSORT(A, ℓ + 1, g − 1)
23 YBBQUICKSORT(A, g + 1, right )
```

YBB Partitioning Example



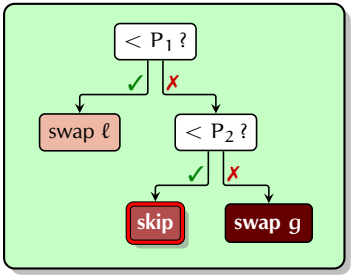


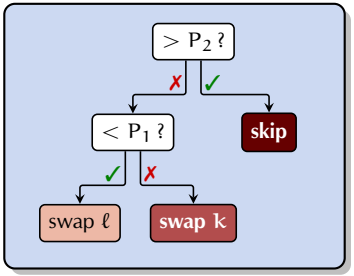


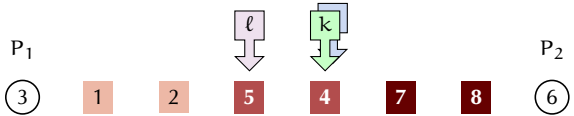


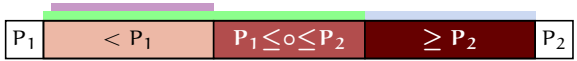

```
YBBQUICKSORT(A, left, right)
1  if right < left then return
2  P1 := A[left];    P2 := A[right]
3  if P1 > P2 then Swap P1 and P2 end if
4  ℓ := left + 1;    g := right − 1;    k := ℓ
5  while k ⩽ g
6      if A[k] < P1
7          Swap A[k] and A[ℓ] ;  ℓ := ℓ + 1
8      else if A[k] ⩾ P2
9          while A[g] > P2 and k < g
10             g := g − 1
11         end while
12         Swap A[k] and A[g] ;  g := g − 1
13         if A[k] < P1
14             Swap A[k] and A[ℓ] ;  ℓ := ℓ + 1
15         end if
16     end if
17     k := k + 1
18 end while
19 ℓ := ℓ − 1;    g := g + 1
20 Swap A[left] and A[ℓ] ; Swap A[right] and A[g]
21 YBBQUICKSORT(A, left , ℓ − 1)
22 YBBQUICKSORT(A, ℓ + 1, g − 1)
23 YBBQUICKSORT(A, g + 1, right )
```

YBB Partitioning Example



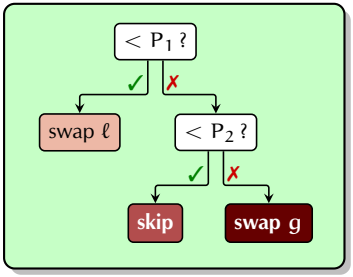


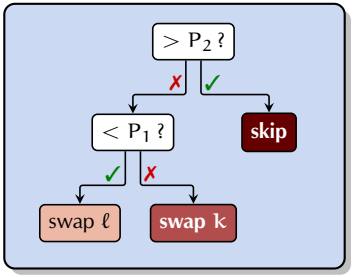





```
YBBQUICKSORT(A, left, right)
1  if right ⩽ left then return
2  P1 := A[left];    P2 := A[right]
3  if P1 > P2 then Swap P1 and P2 end if
4  ℓ := left + 1;    g := right − 1;    k := ℓ
5  while k ⩽ g
6      if A[k] < P1
7          Swap A[k] and A[ℓ] ;  ℓ := ℓ + 1
8      else if A[k] ⩾ P2
9          while A[g] > P2 and k < g
10             g := g − 1
11         end while
12         Swap A[k] and A[g] ;  g := g − 1
13         if A[k] < P1
14             Swap A[k] and A[ℓ] ;  ℓ := ℓ + 1
15         end if
16     end if
17     k := k + 1
18 end while
19 ℓ := ℓ − 1;    g := g + 1
20 Swap A[left] and A[ℓ] ; Swap A[right] and A[g]
21 YBBQUICKSORT(A, left , ℓ − 1)
22 YBBQUICKSORT(A, ℓ + 1, g − 1)
23 YBBQUICKSORT(A, g + 1, right )
```

YBB Partitioning Example



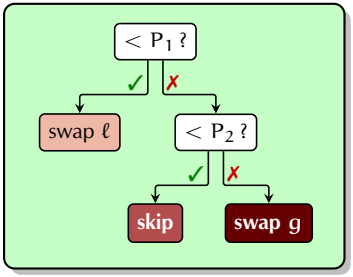


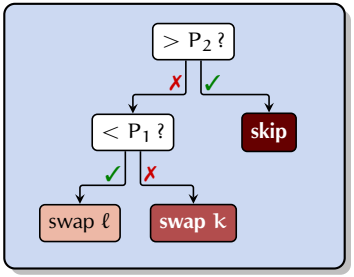


YBBQUICKSORT(A, *left*, *right*)

```
1  if right ⩽ left then return
2  P1 := A[left];    P2 := A[right]
3  if P1 > P2 then Swap P1 and P2 end if
4  ℓ := left + 1;    g := right − 1;    k := ℓ
5  while k ⩽ g
6        if A[k] < P1
7              Swap A[k] and A[ℓ] ;  ℓ := ℓ + 1
8        else if A[k] ⩾ P2
9              while A[g] > P2 and k < g
10                   g := g − 1
11             end while
12             Swap A[k] and A[g] ;   g := g − 1
13             if A[k] < P1
14                   Swap A[k] and A[ℓ] ;   ℓ := ℓ + 1
15             end if
16       end if
17       k := k + 1
18 end while
19 ℓ := ℓ − 1;    g := g + 1
20 Swap A[left] and A[ℓ] ; Swap A[right] and A[g]
21 YBBQUICKSORT(A,  left  , ℓ − 1)
22 YBBQUICKSORT(A, ℓ + 1, g − 1)
23 YBBQUICKSORT(A, g + 1, right )
```

YBB Partitioning Example



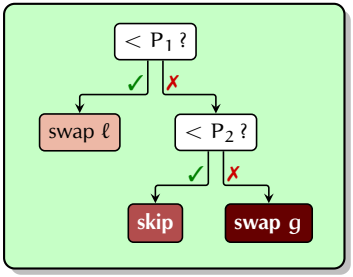


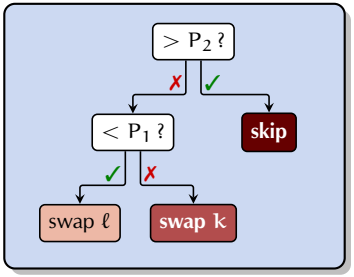


YBBQUICKSORT(A, *left*, *right*)

```
1  if right ⩽ left then return
2  P1 := A[left];    P2 := A[right]
3  if P1 > P2 then Swap P1 and P2 end if
4  ℓ := left + 1;    g := right − 1;    k := ℓ
5  while k ⩽ g
6        if A[k] < P1
7              Swap A[k] and A[ℓ] ;  ℓ := ℓ + 1
8        else if A[k] ⩾ P2
9              while A[g] > P2 and k < g
10                   g := g − 1
11             end while
12             Swap A[k] and A[g] ;   g := g − 1
13             if A[k] < P1
14                   Swap A[k] and A[ℓ] ;   ℓ := ℓ + 1
15             end if
16       end if
17       k := k + 1
18 end while
19 ℓ := ℓ − 1;    g := g + 1
20 Swap A[left] and A[ℓ] ; Swap A[right] and A[g]
21 YBBQUICKSORT(A, left , ℓ − 1)
22 YBBQUICKSORT(A, ℓ + 1, g − 1)
23 YBBQUICKSORT(A, g + 1, right )
```

YBB Partitioning Example





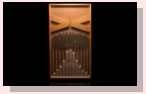


YBBQUICKSORT(A, *left*, *right*)

```
1  if right ⩽ left then return
2  P1 := A[left];    P2 := A[right]
3  if P1 > P2 then Swap P1 and P2 end if
4  ℓ := left + 1;    g := right − 1;    k := ℓ
5  while k ⩽ g
6        if A[k] < P1
7              Swap A[k] and A[ℓ] ;  ℓ := ℓ + 1
8        else if A[k] ⩾ P2
9              while A[g] > P2 and k < g
10                   g := g − 1
11             end while
12             Swap A[k] and A[g] ;   g := g − 1
13             if A[k] < P1
14                   Swap A[k] and A[ℓ] ;   ℓ := ℓ + 1
15             end if
16       end if
17       k := k + 1
18 end while
19 ℓ := ℓ − 1;    g := g + 1
20 Swap A[left] and A[ℓ] ; Swap A[right] and A[g]
21 YBBQUICKSORT(A,  left  , ℓ − 1)
22 YBBQUICKSORT(A, ℓ + 1, g − 1)
23 YBBQUICKSORT(A, g + 1, right )
```

Outline







1 Context

2 Dual-Pivot Quicksort

3 Average-Case Analysis

4 Quicksort + X

5 Conclusion



3 Average-Case Analysis

Comparisons

How many comparisons on average?

- Consider **first** partitioning step.
- How many #cmps per element?
  Well, either 2 or 1.
- How many **on average**?

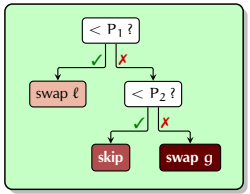
Analysis Sketch:

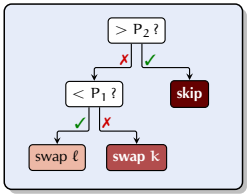
- Have to compare with one pivot first, say P_1.
  (Similar if P_2 used first.)

$\rightsquigarrow$ 1 cmp for small elements,
2 cmps for medium and large.

- On average, $\sim \frac{1}{3}n$ elements each kind.

$\rightsquigarrow$ on average $2 \cdot \frac{2}{3} + 1 \cdot \frac{1}{3} = \frac{5}{3}$ cmps per element

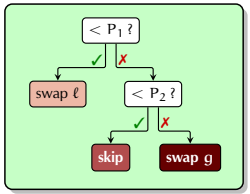


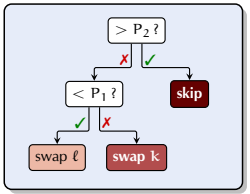


Comparisons

How many comparisons on average?

- Consider **first** partitioning step.
- How many #cmps per element?
  Well, either 2 or 1.
- How many **on average**?





Analysis Sketch:

- Have to compare with one pivot first, say P_1.
  (Similar if P_2 used first.)

$\rightsquigarrow$ 1 cmp for small elements,
2 cmps for medium and large.

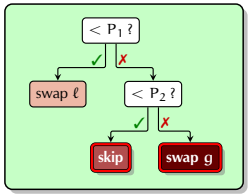
- On average, $\sim \frac{1}{3}n$ elements each kind.

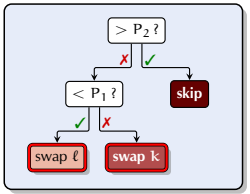
$\rightsquigarrow$ on average $2 \cdot \frac{2}{3} + 1 \cdot \frac{1}{3} = \frac{5}{3}$ cmps per element

Comparisons

How many comparisons on average?

- Consider **first** partitioning step.
- How many #cmps per element?
  Well, either 2 or 1.
- How many **on average**?





Analysis Sketch:

- Have to compare with one pivot first, say P_1.
  (Similar if P_2 used first.)

$\rightsquigarrow$ 1 cmp for small elements,
2 cmps for medium and large.

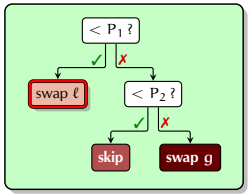
- On average, $\sim \frac{1}{3}n$ elements each kind.

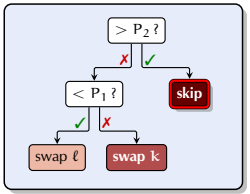
$\rightsquigarrow$ on average $2 \cdot \frac{2}{3} + 1 \cdot \frac{1}{3} = \frac{5}{3}$ cmps per element

Comparisons

How many comparisons on average?

- Consider **first** partitioning step.
- How many #cmps per element?
  Well, either 2 or 1.
- How many **on average**?





Analysis Sketch:

- Have to compare with one pivot first, say P_1.
  (Similar if P_2 used first.)

$\rightsquigarrow$ 1 cmp for small elements,
2 cmps for medium and large.

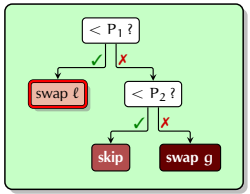
- On average, $\sim \frac{1}{3}n$ elements each kind.

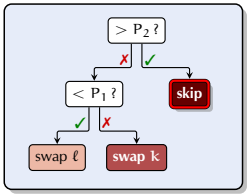
$\rightsquigarrow$ on average $2 \cdot \frac{2}{3} + 1 \cdot \frac{1}{3} = \frac{5}{3}$ cmps per element

Comparisons

How many comparisons on average?

- Consider **first** partitioning step.
- How many #cmps per element?
  Well, either 2 or 1.
- How many **on average**?





Analysis Sketch:

- Have to compare with one pivot first, say P_1.
  (Similar if P_2 used first.)

$\rightsquigarrow$ 1 cmp for small elements,

2 cmps for medium and large.

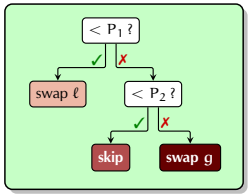
- On average, $\sim \frac{1}{3}n$ elements each kind.

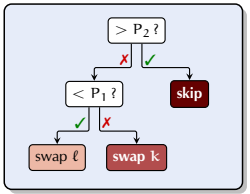
$\rightsquigarrow$ on average $2 \cdot \frac{2}{3} + 1 \cdot \frac{1}{3} = \frac{5}{3}$ cmps per element

Comparisons

How many comparisons on average?

- Consider **first** partitioning step.
- How many #cmps per element?
  Well, either 2 or 1.
- How many **on average**?





Analysis Sketch:

- Have to compare with one pivot first, say P_1.
  (Similar if P_2 used first.)

$\rightsquigarrow$ 1 cmp for small elements,
2 cmps for medium and large.

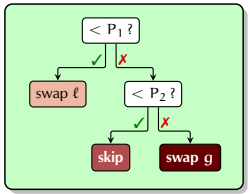
- On average, $\sim \frac{1}{3}n$ elements each kind.

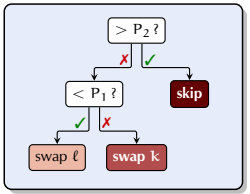
$\rightsquigarrow$ on average $2 \cdot \frac{2}{3} + 1 \cdot \frac{1}{3} = \frac{5}{3}$ cmps per element

Comparisons

How many comparisons on average?

- Consider **first** partitioning step.
- How many #cmps per element?
  Well, either 2 or 1.
- How many **on average**?





Analysis Sketch:

- Have to compare with one pivot first, say P_1.
  (Similar if P_2 used first.)

$\rightsquigarrow$ 1 cmp for small elements,
2 cmps for medium and large.

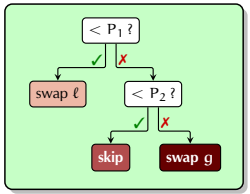
- On average, $\sim \frac{1}{3}n$ elements each kind.

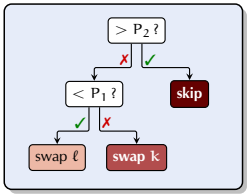
$\rightsquigarrow$ on average $2 \cdot \frac{2}{3} + 1 \cdot \frac{1}{3} = \frac{5}{3}$ cmps per element

Comparisons

How many comparisons on average?

- Consider **first** partitioning step.
- How many #cmps per element?
  Well, either 2 or 1.
- How many **on average**?





Analysis Sketch:

- Have to compare with one pivot first, say P_1.
  (Similar if P_2 used first.)

⇝ 1 cmp for small elements,
2 cmps for medium and large.

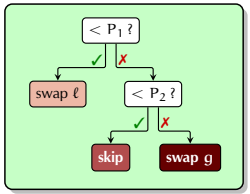
- On average, $\sim \frac{1}{3}n$ elements each kind.

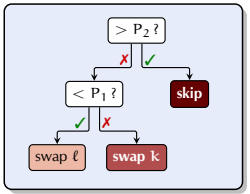
⇝ on average $2 \cdot \frac{2}{3} + 1 \cdot \frac{1}{3} = \frac{5}{3}$ cmps per element

Comparisons

How many comparisons on average?

- Consider **first** partitioning step.
- How many #cmps per element?
  Well, either 2 or 1.
- How many **on average**?





Analysis Sketch:

- Have to compare with one pivot first, say P_1.
  (Similar if P_2 used first.)

⇝ 1 cmp for small elements,
2 cmps for medium and large.

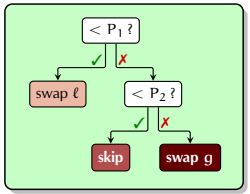
- On average, $\sim \frac{1}{3}n$ elements each kind.

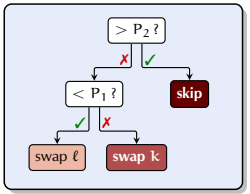
⇝ on average $2 \cdot \frac{2}{3} + 1 \cdot \frac{1}{3} = \frac{5}{3}$ cmps per element

Comparisons

How many comparisons on average?

- Consider **first** partitioning step.
- How many #cmps per element?
  Well, either 2 or 1.
- How many **on average**?





Analysis Sketch:

- Have to compare with one pivot first, say P_1.
  (Similar if P_2 used first.)

⇝ 1 cmp for small elements,
2 cmps for medium and large.

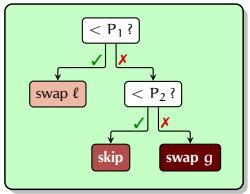
- On average, $\sim \frac{1}{3}n$ elements each kind.

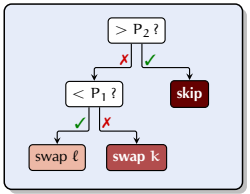
⇝ on average $2 \cdot \frac{2}{3} + 1 \cdot \frac{1}{3} = \frac{5}{3}$ cmps per element

Comparisons

How many comparisons on average?

- Consider **first** partitioning step.
- How many #cmps per element?
  Well, either 2 or 1.
- How many **on average**?





Analysis Sketch:

- Have to compare with one pivot first, say P_1.
  (Similar if P_2 used first.)

$\rightsquigarrow$ 1 cmp for small elements,
2 cmps for medium and large.

- On average, $\sim \frac{1}{3}n$ elements each kind.

$\rightsquigarrow$ on average $2 \cdot \frac{2}{3} + 1 \cdot \frac{1}{3} = \frac{5}{3}$ cmps per element

- Sounds plausible?

Comparisons

How many comparisons on average?

- Consider **first** partitioning step.
- How many #cmps per element?
  Well, either 2 or 1.
- How many **on average**?

Analysis Sketch:

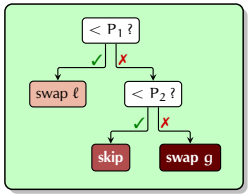
- Have to compare with one pivot first, say P_1.
  (Similar if P_2 used first.)

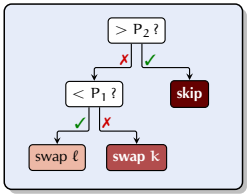
⇝ 1 cmp for small elements,
2 cmps for medium and large.

- On average, $\sim \frac{1}{3}n$ elements each kind.

⇝ on average $2 \cdot \frac{2}{3} + 1 \cdot \frac{1}{3} = \frac{5}{3}$ cmps per element

- Sounds plausible? . . . but it's wrong!





Comparisons

How many comparisons on average?

- Consider **first** partitioning step.
- How many #cmps per element?
  Well, either 2 or 1.
- How many **on average**?

Analysis Sketch:

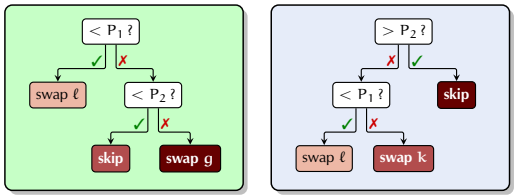
- Have to compare with one pivot first, say P_1.
  (Similar if P_2 used first.)

$\rightsquigarrow$ 1 cmp for small elements,
2 cmps for medium and large.

- On average, $\sim \frac{1}{3}n$ elements each kind.

$\rightsquigarrow$ on average $2 \cdot \frac{2}{3} + 1 \cdot \frac{1}{3} = \frac{5}{3}$ cmps per element

- Sounds plausible? . . . but it's wrong!



Problem: There are **correlations**.

- many small elements $\rightsquigarrow$ k's range big
- many large elements $\rightsquigarrow$ g's range big

$\rightsquigarrow$ $> \frac{1}{3}n$ cheap elements on average.

How many? . . . Analysis of Algorithms to the rescue!

Comparisons

How many comparisons on average?

- Consider **first** partitioning step.
- How many #cmps per element?
  Well, either 2 or 1.
- How many **on average**?

Analysis Sketch:

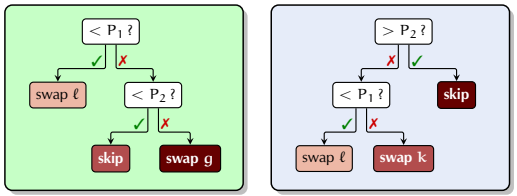
- Have to compare with one pivot first, say P_1.
  (Similar if P_2 used first.)

$\rightsquigarrow$ 1 cmp for small elements,
2 cmps for medium and large.

- On average, $\sim \frac{1}{3}n$ elements each kind.

$\rightsquigarrow$ on average $2 \cdot \frac{2}{3} + 1 \cdot \frac{1}{3} = \frac{5}{3}$ cmps per element

- Sounds plausible? . . . but it's wrong!



Problem: There are **correlations**.

- many small elements $\rightsquigarrow$ k's range big
- many large elements $\rightsquigarrow$ g's range big

$\rightsquigarrow > \frac{1}{3}n$ cheap elements on average.

How many? . . . Analysis of Algorithms to the rescue!

Comparisons

How many comparisons on average?

- Consider **first** partitioning step.
- How many #cmps per element?
  Well, either 2 or 1.
- How many **on average**?

Analysis Sketch:

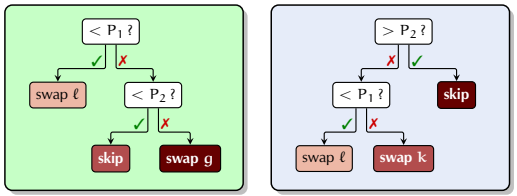
- Have to compare with one pivot first, say P_1.
  (Similar if P_2 used first.)

$\rightsquigarrow$ 1 cmp for small elements,
2 cmps for medium and large.

- On average, $\sim \frac{1}{3}n$ elements each kind.

$\rightsquigarrow$ on average $2 \cdot \frac{2}{3} + 1 \cdot \frac{1}{3} = \frac{5}{3}$ cmps per element

- Sounds plausible? . . . but it's wrong!



Problem: There are **correlations**.

- many small elements $\rightsquigarrow$ k's range big
- many large elements $\rightsquigarrow$ g's range big

$\rightsquigarrow > \frac{1}{3}n$ cheap elements on average.

How many? . . . Analysis of Algorithms to the rescue!

Comparisons

How many comparisons on average?

- Consider **first** partitioning step.
- How many #cmps per element?
  Well, either 2 or 1.
- How many **on average**?

Analysis Sketch:

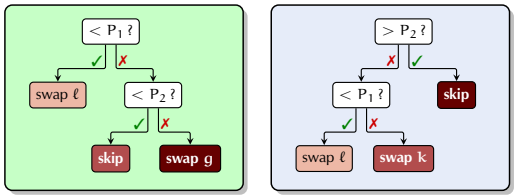
- Have to compare with one pivot first, say P_1.
  (Similar if P_2 used first.)

$\rightsquigarrow$ 1 cmp for small elements,
2 cmps for medium and large.

- On average, $\sim \frac{1}{3}n$ elements each kind.

$\rightsquigarrow$ on average $2 \cdot \frac{2}{3} + 1 \cdot \frac{1}{3} = \frac{5}{3}$ cmps per element

- Sounds plausible? . . . but it's wrong!



Problem: There are **correlations**.

- many small elements $\rightsquigarrow$ k's range big
- many large elements $\rightsquigarrow$ g's range big

$\rightsquigarrow$ $> \frac{1}{3}n$ cheap elements on average.

How many? . . . Analysis of Algorithms to the rescue!

Comparisons

How many comparisons on average?

- Consider **first** partitioning step.
- How many #cmps per element?
  Well, either 2 or 1.
- How many **on average**?

Analysis Sketch:

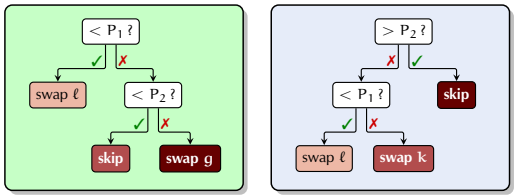
- Have to compare with one pivot first, say P_1.
  (Similar if P_2 used first.)

$\rightsquigarrow$ 1 cmp for small elements,
2 cmps for medium and large.

- On average, $\sim \frac{1}{3}n$ elements each kind.

$\rightsquigarrow$ on average $2 \cdot \frac{2}{3} + 1 \cdot \frac{1}{3} = \frac{5}{3}$ cmps per element

- Sounds plausible? . . . but it's wrong!



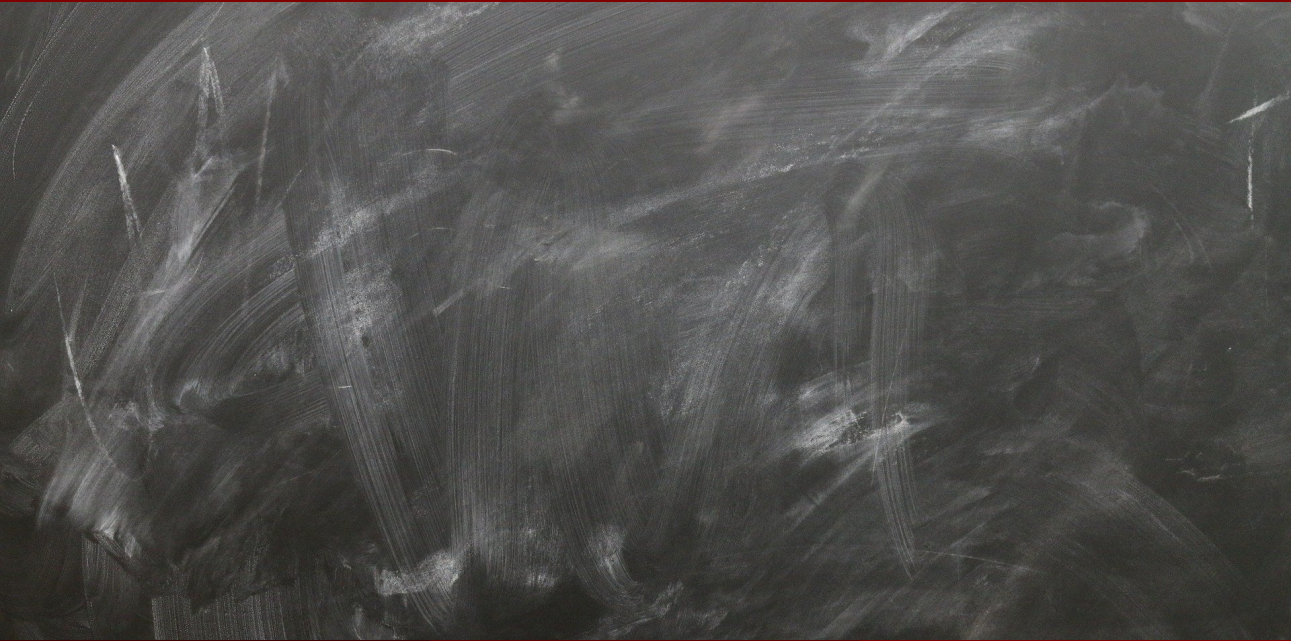
Problem: There are **correlations**.

- many small elements $\rightsquigarrow$ k's range big
- many large elements $\rightsquigarrow$ g's range big

$\rightsquigarrow$ $> \frac{1}{3}n$ cheap elements on average.

How many? . . . Analysis of Algorithms to the rescue!

Mathematical Analysis of Algorithms



Mathematical Analysis of Algorithms

Random Model

- n **i.i.d.** elements chosen **uniformly** in $(0,1)$



  - pairwise **distinct** almost surely
  - relative ranking is a **random permutation**
  - $\rightsquigarrow$ equivalent to classic model

- For **dual-pivot** Quicksort with **fixed** pivots $P_1 \leqslant P_2$
  - $\Pr[\ U_i < P_1\ ] = D_1$
  - $\Pr[\ P_1 < U_i < P_2\ ] = D_2$
  - $\Pr[\ U_i > P_2\ ] = D_3$

$\vec{D}$ is a uniformly chosen probability distribution!



- Known distribution: $\vec{D} = (D_1, D_2, D_3] \overset{d}{=} \text{Dirichlet}(1,1,1)$.
  $(D_1, D_2]$ uniform in 2D simplex $\{x, y \in (0,1) : x + y < 1\}$



Mathematical Analysis of Algorithms

Random Model

- n **i.i.d.** elements chosen **uniformly** in $(0,1)$



  - pairwise **distinct** almost surely
  - relative ranking is a **random permutation**
  - $\rightsquigarrow$ equivalent to classic model

- For **dual-pivot** Quicksort with **fixed** pivots $P_1 \leqslant P_2$
  - $\Pr[\ U < P_1\ ] = D_1$
  - $\Pr[\ P_1 < U < P_2\ ] = D_2$
  - $\Pr[\ U > P_2\ ] = D_3$



$\vec{D}$ is a uniformly chosen probability distribution!

- Known distribution: $\vec{D} = (D_1, D_2, D_3) \triangleq \text{Dirichlet}(1,1,1)$.
  (D_1, D_2) uniform in 2D simplex $\{x, y \in (0,1) : x + y < 1\}$



Comparisons in YBB Partitioning

- $t(n)$: $\mathbb{E}[\#\text{cmps}]$ in **first partitioning step**



- $I_{\geqslant P_2}$: number of small elements
- $I_{\geqslant P_2}$: number of medium elements
- $I_{\geqslant P_2}$: number of large elements

To be precise, we have **conditional** on $\vec{I}$:

$I_{\geqslant S} \triangleq \text{HypG}(n-2, \frac{I_1}{n}, I_3)$

$\rightsquigarrow \mathbb{E}[I_{\geqslant S} \mid \vec{I}] = \frac{I_1}{n-2}$

Draw positions of large elements

$I_{\geqslant S} = 1$

$\rightsquigarrow t(n) = \mathbb{E}[$

$\vec{I} = (I_1, I_2, I_3) \triangleq \text{Mult}(n-2, \vec{D})$

$\rightsquigarrow \vec{I} = \vec{D}n \pm O(\sqrt{n})$ whp

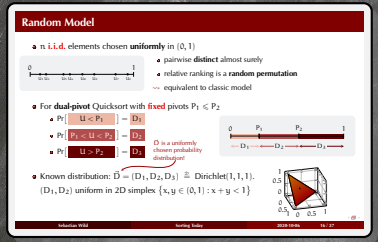
Fraction of elements in $\mathbb{X}$

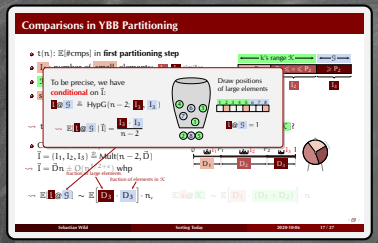


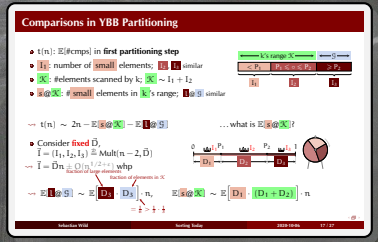
$\rightsquigarrow \mathbb{E}[I_{\geqslant S}] \sim \mathbb{E}[D_1 \cdot D_3] \cdot n,$ $\mathbb{E}[\#\mathbb{X}] \sim \mathbb{E}[D_1 \cdot (D_1 + D_2)] \cdot n$

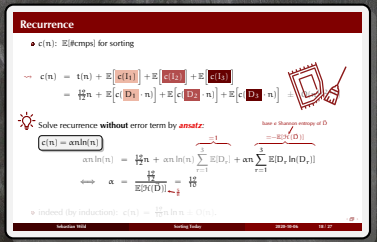
Mathematical Analysis of Algorithms

Mathematical Analysis of Algorithms

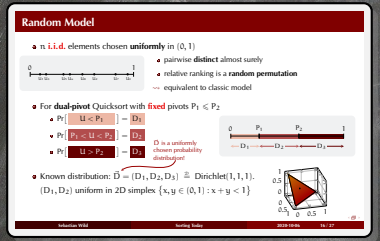


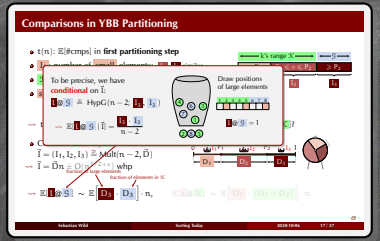


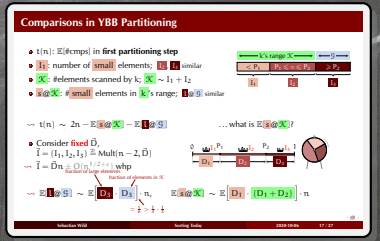


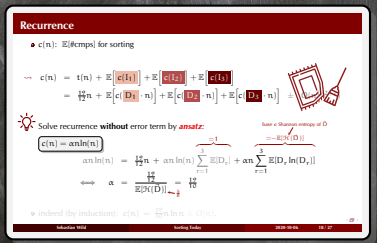


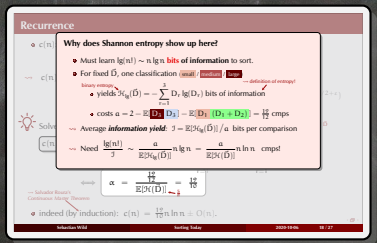
Mathematical Analysis of Algorithms



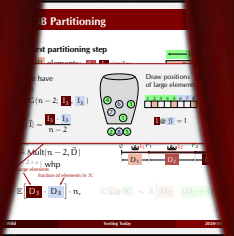








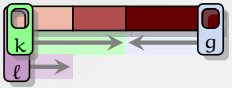
Mathematical Analysis of Algorithms



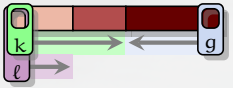
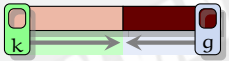
Time for a Competition



Time for a Competition

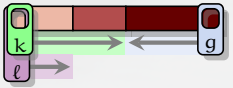
	Dual-Pivot Quicksort	Standard Quicksort	Relative Difference	Reference
				

Time for a Competition

	Dual-Pivot Quicksort	Standard Quicksort	Relative Difference	Reference
				
comparisons	1.9	2	−5%	ESA 2012

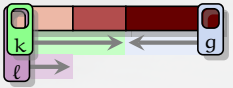
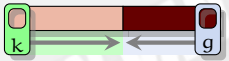
$\cdot n \ln n + O(n)$ *average case results*

Time for a Competition

	Dual-Pivot Quicksort	Standard Quicksort	Relative Difference	Reference
				
comparisons	1.9	2	−5%	ESA 2012
write accesses	$\frac{11}{10} \approx 1.1$	$\frac{2}{3} \approx 0.\overline{6}$	+65%	ESA 2012

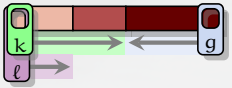
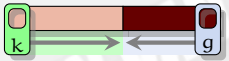
$\cdot n \ln n + O(n)$ *average case results*

Time for a Competition

	Dual-Pivot Quicksort	Standard Quicksort	Relative Difference	Reference
				
comparisons	1.9	2	−5%	ESA 2012
write accesses	$\frac{11}{10} \approx 1.1$	$\frac{2}{3} \approx 0.\overline{6}$	+65%	ESA 2012
Java Bytecode instr.	23.8	18	+32.2%	TALG 2015

$\cdot n \ln n + O(n)$ *average case results*

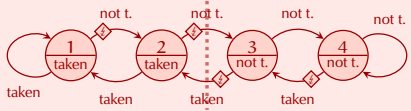
Time for a Competition

	Dual-Pivot Quicksort	Standard Quicksort	Relative Difference	Reference
				
comparisons	1.9	2	−5%	ESA 2012
write accesses	$\frac{11}{10} \approx 1.1$	$\frac{2}{3} \approx 0.\overline{6}$	+65%	ESA 2012
Java Bytecode instr.	23.8	18	+32.2%	TALG 2015
branch mispredictions (2-bit saturating counter)				

$\cdot n \ln n + O(n)$ *average case results*

Time for a Competition

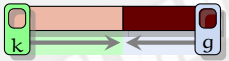
CPU tries to **predict** whether a **conditional branch** will be taken



Prediction wrong ↯ ⇝ *Pipeline Stall*

Java Bytecode instr. 23.8

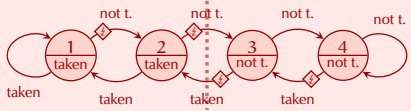
branch mispredictions
(2-bit saturating counter)

Standard Quicksort	Relative Difference	Reference
		
2	−5%	ESA 2012
$\frac{2}{3} \approx 0.\overline{6}$	+65%	ESA 2012
18	+32.2%	TALG 2015

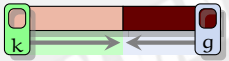
$\cdot n \ln n + O(n)$ *average case results*

Time for a Competition

CPU tries to **predict** whether a **conditional branch** will be taken

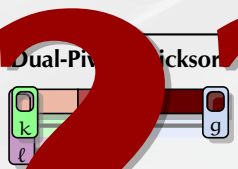


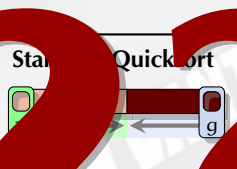
Prediction wrong ↯ ⇝ *Pipeline Stall*

		Standard Quicksort	Relative Difference	Reference
				
		2	−5%	ESA 2012
		$\frac{2}{3} \approx 0.\overline{6}$	+65%	ESA 2012
Java Bytecode instr.	23.8	18	+32.2%	TALG 2015
branch mispredictions (2-bit saturating counter)	$\frac{31\pi}{40} - \frac{37}{20} \approx 0.5847$	$\frac{\pi}{2} - 1 \approx 0.5708$	+2.4%	ANALCO 2015

$\cdot n \ln n + O(n)$ *average case results*

Time for a Competition





	Dual-Piv…	icksor…	Sta…	Quick…ort	…tive …ence	Reference
comparisons	1.9				−5%	ESA 2012
write accesses	$\frac{11}{10} \approx 1.1$			$\frac{2}{3} \approx 0.\overline{6}$		ESA 2012
Java Bytecode instr.	23.8			18	+32.2%	TALG 2015
branch mispredictions (2-bit saturating counter)	$\frac{3}{4} \approx 0.5847$			$1 \approx 0.5708$		ANALCO 2015

$\cdot n \ln n + O(n)$ *average case results*

What happened?

What do comparisons, write accesses, Bytecodes and branch misses have in common?

$\rightsquigarrow$ they happen **inside** the **CPU**

What happened?

What do comparisons, write accesses, Bytecodes and branch misses have in common?

$\rightsquigarrow$ they happen **inside** the **CPU**

What happened?

What do comparisons, write accesses, Bytecodes and branch misses have in common?

$\rightsquigarrow$ they happen **inside** the **CPU** ...but we need to get the data there, as well!

What happened?

What do comparisons, write accesses, Bytecodes and branch misses have in common?

$\rightsquigarrow$ they happen **inside** the **CPU** ...but we need to get the data there, as well!

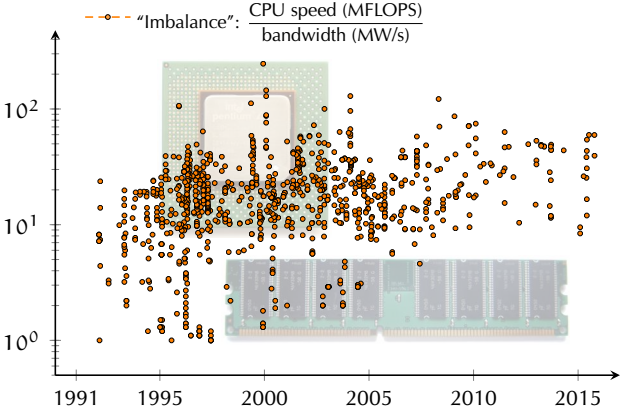
Average annual growth rates:

CPU speed:	46%
Memory Bandwidth:	37%

What happened?

What do comparisons, write accesses, Bytecodes and branch misses have in common?

$\rightsquigarrow$ they happen **inside** the **CPU** ...but we need to get the data there, as well!



Average annual growth rates:

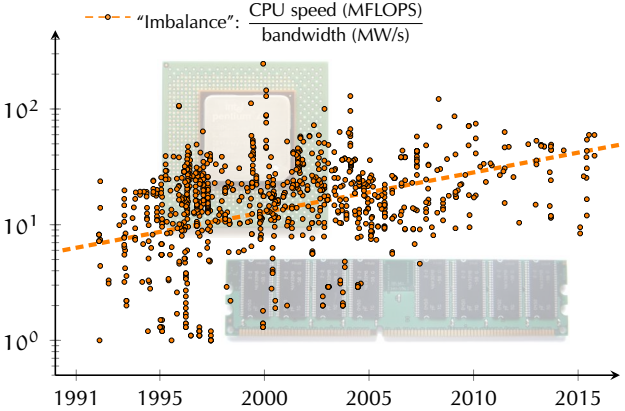
CPU speed:	46%
Memory Bandwidth:	37%

STREAM Benchmark data with linear regression
www.cs.virginia.edu/stream/by_date/Balance.html

What happened?

What do comparisons, write accesses, Bytecodes and branch misses have in common?

$\rightsquigarrow$ they happen **inside** the **CPU** ...but we need to get the data there, as well!



STREAM Benchmark data with linear regression
www.cs.virginia.edu/stream/by_date/Balance.html

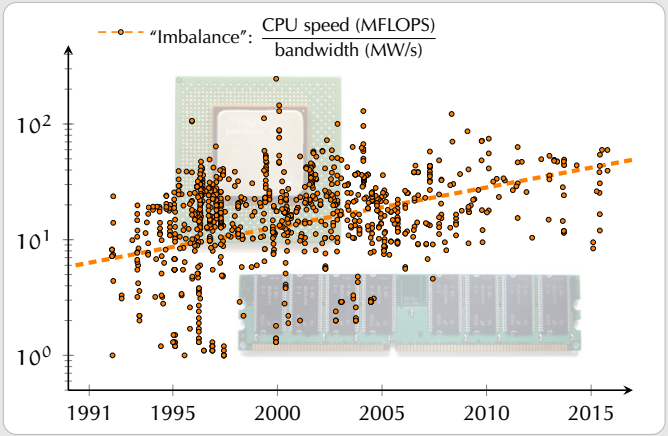
Average annual growth rates:

CPU speed:	46%
Memory Bandwidth:	37%
$\rightsquigarrow$ Imbalance:	5%

What happened?

What do comparisons, write accesses, Bytecodes and branch misses have in common?

⇝ they happen **inside** the **CPU** …but we need to get the data there, as well!



Average annual growth rates:

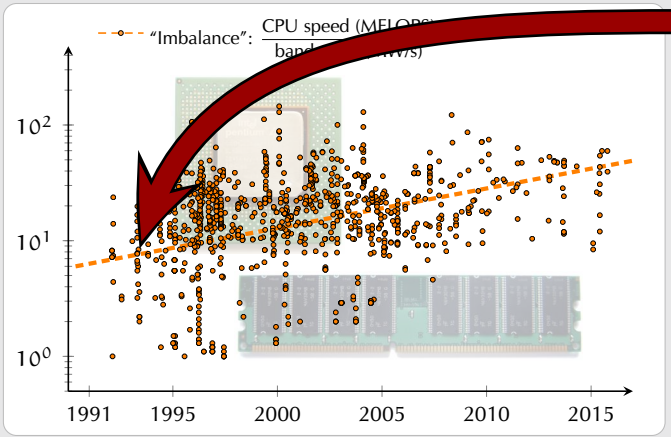
CPU speed:	46%
Memory Bandwidth:	37%
⇝ Imbalance:	5%

STREAM Benchmark data with linear regression
www.cs.virginia.edu/stream/by_date/Balance.html

What happened?

What do comparisons, write accesses, Bytecodes and branc

⇝ they happen **inside** the **CPU** …but we need to get t



Classic Quicksort (Bentley & McIlroy) **1993**

- 1 mem transfer ≈ 7 instructions

⇝ focus on CPU costs

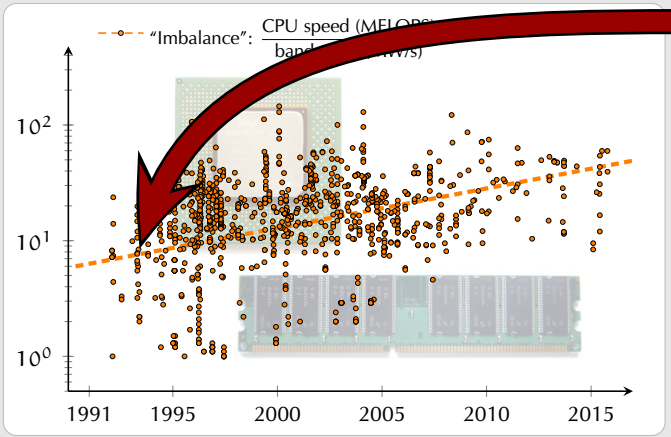
CPU speed:	46%
Memory Bandwidth:	37%
⇝ Imbalance:	5%

STREAM Benchmark data with linear regression
www.cs.virginia.edu/stream/by_date/Balance.html

What happened?

What do comparisons, write accesses, Bytecodes and branc

⇝ they happen **inside** the **CPU** …but we need to get t



Classic Quicksort (Bentley & McIlroy) **1993**

- 1 mem transfer ≈ 7 instructions

⇝ focus on CPU costs

CPU speed: 46%

Memory Bandwidth: 37%

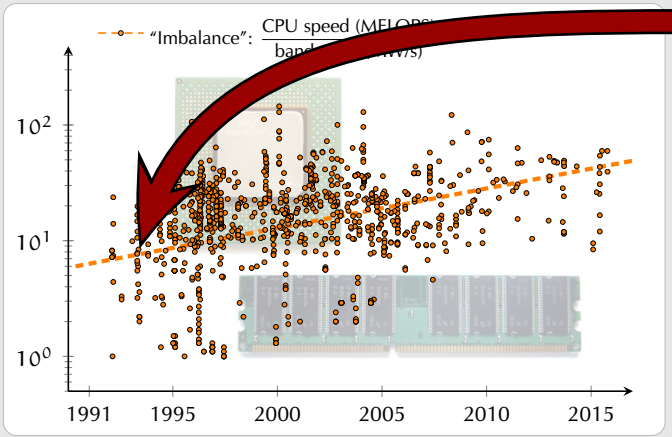
⇝ Imbalance: 5%

STREAM Benchmark data with linear regression
www.cs.virginia.edu/stream/by_date/Balance.html

What happened?

What do comparisons, write accesses, Bytecodes and branc

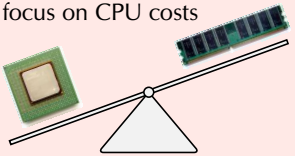
⇝ they happen **inside** the **CPU** …but we need to get t



Classic Quicksort (Bentley & McIlroy) **1993**

- 1 mem transfer ≈ 7 instructions

⇝ focus on CPU costs



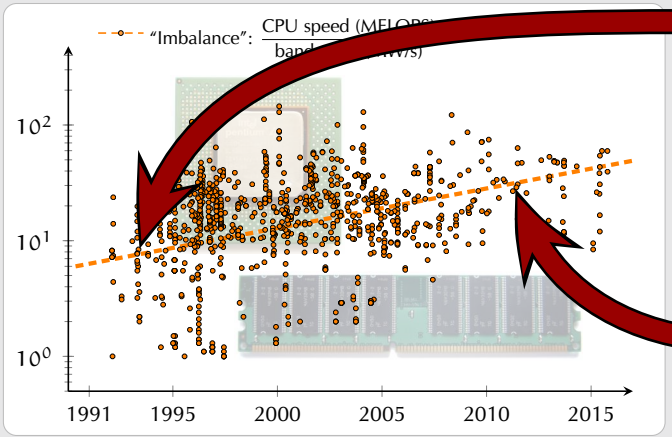
CPU speed:	46%
Memory Bandwidth:	37%
⇝ Imbalance:	5%

STREAM Benchmark data with linear regression
www.cs.virginia.edu/stream/by_date/Balance.html

What happened?

What do comparisons, write accesses, Bytecodes and branc

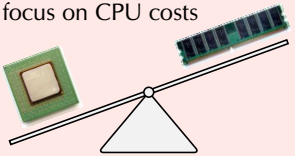
⇝ they happen **inside** the **CPU** …but we need to get t



Classic Quicksort (Bentley & McIlroy) **1993**

- 1 mem transfer ≈ 7 instructions

⇝ focus on CPU costs



YBB Quicksort, **2009 – today**

- 1 transfer ≈ 30 instructions

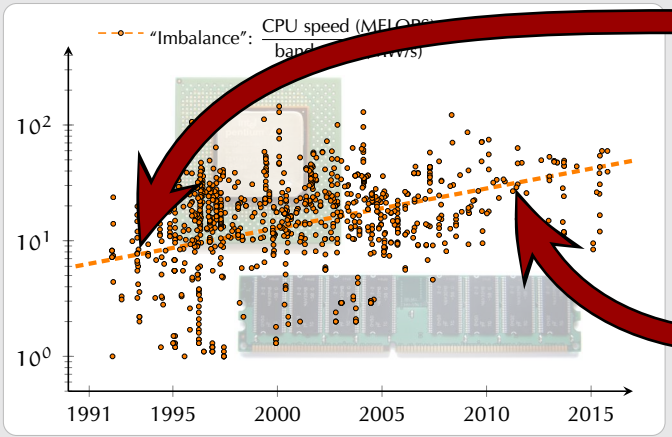
⇝ focus on CPU and bandwidth

STREAM Benchmark data with linear regression
www.cs.virginia.edu/stream/by_date/Balance.html

What happened?

What do comparisons, write accesses, Bytecodes and branc

⇝ they happen **inside** the **CPU** …but we need to get t

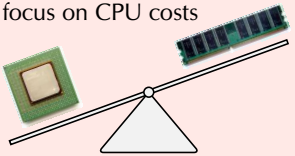


STREAM Benchmark data with linear regression
www.cs.virginia.edu/stream/by_date/Balance.html

Classic Quicksort (Bentley & McIlroy) **1993**

- 1 mem transfer ≈ 7 instructions

⇝ focus on CPU costs



YBB Quicksort, **2009 – today**

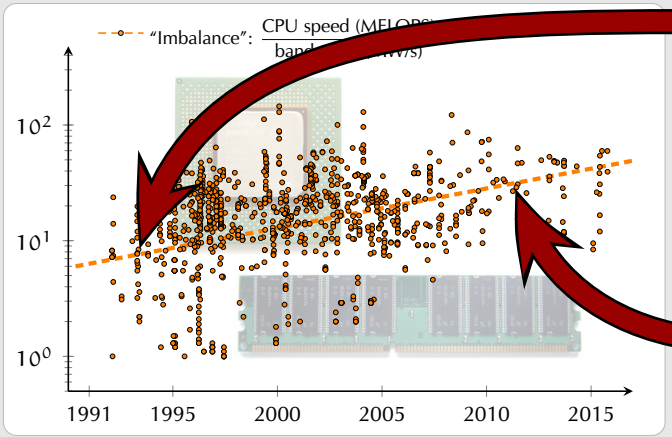
- 1 transfer ≈ 30 instructions

⇝ focus on CPU and bandwidth

What happened?

What do comparisons, write accesses, Bytecodes and branc

⇝ they happen **inside** the **CPU** …but we need to get t

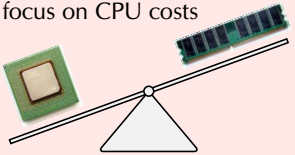


STREAM Benchmark data with linear regression
www.cs.virginia.edu/stream/by_date/Balance.html

Classic Quicksort (Bentley & McIlroy) **1993**

- 1 mem transfer ≈ 7 instructions

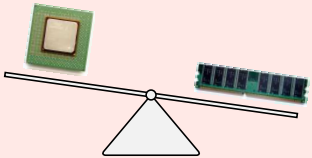
⇝ focus on CPU costs



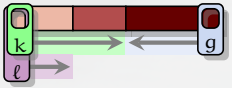
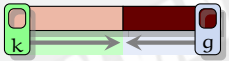
YBB Quicksort, **2009 – today**

- 1 transfer ≈ 30 instructions

⇝ focus on CPU **and bandwidth**

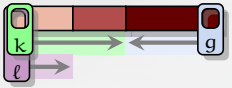
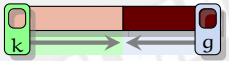


Time for a Competition

	Dual-Pivot Quicksort	Standard Quicksort	Relative Difference	Reference
				
comparisons	1.9	2	−5%	ESA 2012
write accesses	$\frac{11}{10} \approx 1.1$	$\frac{2}{3} \approx 0.\overline{6}$	+65%	ESA 2012
Java Bytecode instr.	23.8	18	+32.2%	TALG 2015
branch mispredictions (2-bit saturating counter)	$\frac{31\pi}{40} - \frac{37}{20} \approx 0.5847$	$\frac{\pi}{2} - 1 \approx 0.5708$	+2.4%	ANALCO 2015

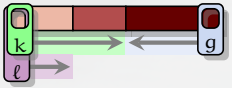
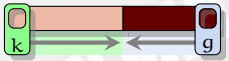
$\cdot n \ln n + O(n)$ *average case results*

Time for a Competition

	Dual-Pivot Quicksort	Standard Quicksort	Relative Difference	Reference
				
comparisons	1.9	2	−5%	ESA 2012
write accesses	$\frac{11}{10} \approx 1.1$	$\frac{2}{3} \approx 0.\overline{6}$	+65%	ESA 2012
Java Bytecode instr.	23.8	18	+32.2%	TALG 2015
branch mispredictions (2-bit saturating counter)	$\frac{31\pi}{40} - \frac{37}{20} \approx 0.5847$	$\frac{\pi}{2} - 1 \approx 0.5708$	+2.4%	ANALCO 2015
scanned elements (∝ memory transfers / IOs)				

$\cdot n \ln n + O(n)$ *average case results*

Time for a Competition

	Dual-Pivot Quicksort	Standard Quicksort	Relative Difference	Reference
				
comparisons	1.9	2	−5%	ESA 2012
write accesses	$\frac{11}{10} \approx 1.1$	$\frac{2}{3} \approx 0.\overline{6}$	+65%	ESA 2012
Java Bytecode instr.	23.8	18	+32.2%	TALG 2015
branch mispredictions (2-bit saturating counter)	$\frac{31\pi}{40} - \frac{37}{20} \approx 0.5847$	$\frac{\pi}{2} - 1 \approx 0.5708$	+2.4%	ANALCO 2015
scanned elements ($\propto$ memory transfers / IOs)	1.6	2	−20%	Algorithmica 2016

$\cdot n \ln n + O(n)$ *average case results*

Time for a Competition

Summary:

				Reference
compari…				ESA 2012
write accesses	$\frac{11}{10} \approx 1.1$	$\frac{2}{3} \approx 0.\overline{6}$	+65%	ESA 2012
Java Bytecode instr.	23.8	18	+32.2%	TALG 2015
branch mispredictions (2-bit saturating counter)	$\frac{31\pi}{40} - \frac{37}{20} \approx 0.5847$	$\frac{\pi}{2} - 1 \approx 0.5708$	+2.4%	ANALCO 2015
scanned elements ($\propto$ memory transfers / IOs)	1.6	2	−20%	Algorithmica 2016

$\cdot n \ln n + O(n)$ *average case results*

Time for a Competition

Summary:

1. Only convincing explanation for running times: **memory transfers**

				Reference
compari…				ESA 2012
write accesses	$\frac{11}{10} \approx 1.1$	$\frac{2}{3} \approx 0.\overline{6}$	+65%	ESA 2012
Java Bytecode instr.	23.8	18	+32.2%	TALG 2015
branch mispredictions (2-bit saturating counter)	$\frac{31\pi}{40} - \frac{37}{20} \approx 0.5847$	$\frac{\pi}{2} - 1 \approx 0.5708$	+2.4%	ANALCO 2015
scanned elements ($\propto$ memory transfers / IOs)	1.6	2	−20%	Algorithmica 2016

$\cdot n \ln n + O(n)$ *average case results*

Time for a Competition

Summary:

1. Only convincing explanation for running times: **memory transfers**
2. Why was dual-pivot Quicksort not used earlier?

				Reference
comparis…				ESA 2012
write accesses	$\frac{11}{10} \approx 1.1$	$\frac{2}{3} \approx 0.\overline{6}$	+65%	ESA 2012
Java Bytecode instr.	23.8	18	+32.2%	TALG 2015
branch mispredictions (2-bit saturating counter)	$\frac{31\pi}{40} - \frac{37}{20} \approx 0.5847$	$\frac{\pi}{2} - 1 \approx 0.5708$	+2.4%	ANALCO 2015
scanned elements ($\propto$ memory transfers / IOs)	1.6	2	−20%	Algorithmica 2016

$\cdot n \ln n + O(n)$ *average case results*

Time for a Competition

Summary:

1. Only convincing explanation for running times: **memory transfers**
2. Why was dual-pivot Quicksort not used earlier? *Because it was **not faster** then!*

				Reference
comparis…				ESA 2012
write accesses	$\frac{11}{10} \approx 1.1$	$\frac{2}{3} \approx 0.\overline{6}$	+65%	ESA 2012
Java Bytecode instr.	23.8	18	+32.2%	TALG 2015
branch mispredictions (2-bit saturating counter)	$\frac{31\pi}{40} - \frac{37}{20} \approx 0.5847$	$\frac{\pi}{2} - 1 \approx 0.5708$	+2.4%	ANALCO 2015
scanned elements ($\propto$ memory transfers / IOs)	1.6	2	−20%	Algorithmica 2016

$\cdot n \ln n + O(n)$ *average case results*

Beyond Dual-Pivot: Multiway Partitioning

Can we do even better than YBB Quicksort?

Setup: *Generic s-way one-pass partitioning*

- **s-way partitioning:** split input into $s \geqslant 2$ parts
- **one pass** over input
- m segments grow from left, $s - m$ from right

- classify elements using BSTs: λ_k and λ_g
- pivots $P_1, \ldots, P_{s-1}$ chosen from sample

Beyond Dual-Pivot: Multiway Partitioning

Can we do even better than YBB Quicksort?

Setup: *Generic s-way one-pass partitioning*

- **s-way partitioning:** split input into $s \geqslant 2$ parts
- **one pass** over input
- m segments grow from left, $s - m$ from right

- classify elements using BSTs: λ_k and λ_g
- pivots $P_1, \ldots, P_{s-1}$ chosen from sample

Beyond Dual-Pivot: Multiway Partitioning

Can we do even better than YBB Quicksort?

Setup: *Generic s-way one-pass partitioning*

- **s-way partitioning:** split input into $s \geqslant 2$ parts
- **one pass** over input
- m segments grow from left, $s - m$ from right

- classify elements using BSTs: λ_k and λ_g
- pivots $P_1, \ldots, P_{s-1}$ chosen from sample

Beyond Dual-Pivot: Multiway Partitioning

Can we do even better than YBB Quicksort?

Setup: *Generic s-way one-pass partitioning*

- **s-way partitioning:** split input into $s \geqslant 2$ parts
- **one pass** over input
- m segments grow from left, $s - m$ from right

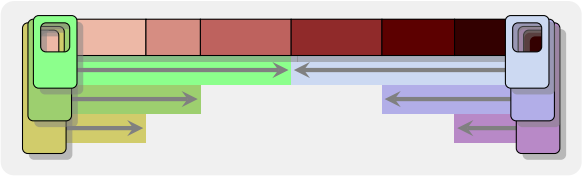
- classify elements using BSTs: λ_k and λ_g
- pivots $P_1, \ldots, P_{s-1}$ chosen from sample

Beyond Dual-Pivot: Multiway Partitioning

Can we do even better than YBB Quicksort?

Setup: *Generic s-way one-pass partitioning*

- **s-way partitioning:** split input into $s \geqslant 2$ parts
- **one pass** over input
- m segments grow from left, $s - m$ from right



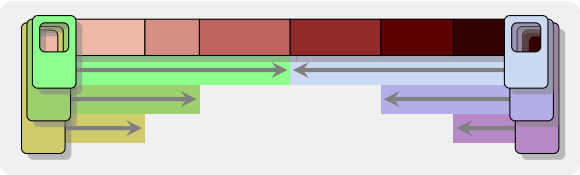
- classify elements using BSTs: λ_k and λ_g
- pivots $P_1, \ldots, P_{s-1}$ chosen from sample

Beyond Dual-Pivot: Multiway Partitioning

Can we do even better than YBB Quicksort?

Setup: *Generic s-way one-pass partitioning*

- **s-way partitioning:** split input into $s \geqslant 2$ parts
- **one pass** over input
- m segments grow from left, $s - m$ from right



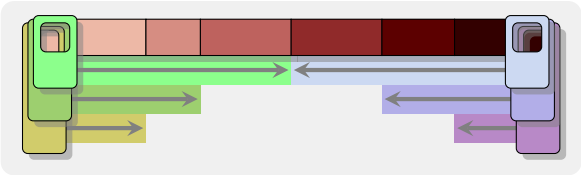
- classify elements using BSTs: λ_k and λ_g
- pivots $P_1, \ldots, P_{s-1}$ chosen from sample

Beyond Dual-Pivot: Multiway Partitioning

Can we do even better than YBB Quicksort?

Setup: *Generic s-way one-pass partitioning*

- **s-way partitioning:** split input into $s \geqslant 2$ parts
- **one pass** over input
- m segments grow from left, $s - m$ from right



- classify elements using BSTs: λ_k and λ_g
- pivots $P_1, \ldots, P_{s-1}$ chosen from sample

Known Quicksort variants

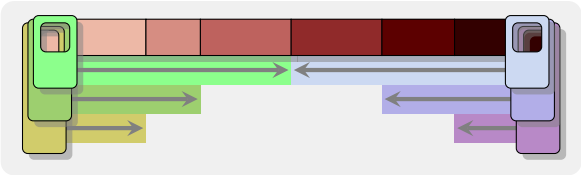
Method	s	m	λ_k	λ_g
Classic	2	1		
Lomuto	2	2		
YBB	3	2		
Waterloo	4	2		

Beyond Dual-Pivot: Multiway Partitioning

Can we do even better than YBB Quicksort?

Setup: *Generic s-way one-pass partitioning*

- **s-way partitioning:** split input into $s \geqslant 2$ parts
- **one pass** over input
- m segments grow from left, $s - m$ from right



- classify elements using BSTs: λ_k and λ_g
- pivots $P_1, \ldots, P_{s-1}$ chosen from sample

Known Quicksort variants

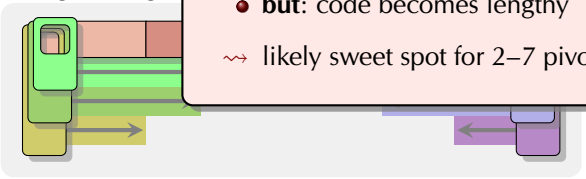
Method	s	m	λ_k	λ_g
Classic	2	1		
Lomuto	2	2		
YBB	3	2		
Waterloo	4	2		

Beyond Dual-Pivot: Multiway Partitioning

Can we do even better than YBB Quicksort?

Setup: *Generic s-way*

- **s-way partitioni**
- **one pass** over in
- m segments gro



- classify elements using BSTs: λ_k and λ_g
- pivots $P_1, \ldots, P_{s-1}$ chosen from sample

Findings for dual-pivot Quicksort are typical:

- **comparisons**: negligible improvements
- **scanned elements**: **significant** improvements
- **but**: code becomes lengthy

⇝ likely sweet spot for 2–7 pivots

…n Quicksort variants

	s	m	λ_k	λ_g
	2	1		
	2	2		
	3	2		
Waterloo	4	2		

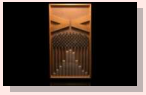
Outline



1 Context



2 Dual-Pivot Quicksort



3 Average-Case Analysis



4 Quicksort + X



5 Conclusion



4 Quicksort + X

What's Wrong with Quicksort?

think: `ints`

- When sorting **simple** objects
  1. **quicksort hard to beat**!
  2. memory transfers a substantial fraction of cost

- What if key **comparisons** are **expensive**?

  ⇝ Quicksort is a factor > 1 off from optimal

  - Heapsort? ↯ cache misses
  - Mergesort? ↯ linear-size buffer needed (or slow)

What's Wrong with Quicksort?

think: `ints`

- When sorting **simple** objects
  1. **quicksort hard to beat**!
  2. memory transfers a substantial fraction of cost

- What if key **comparisons** are **expensive**?

  ⇝ Quicksort is a factor > 1 off from optimal

  - Heapsort? ↯ cache misses
  - Mergesort? ↯ linear-size buffer needed (or slow)

What's Wrong with Quicksort?

think: `ints`

- When sorting **simple** objects
  1. **quicksort hard to beat**!
  2. memory transfers a substantial fraction of cost

- What if key **comparisons** are **expensive**?
  - ⇝ Quicksort is a factor > 1 off from optimal
    - Heapsort? ↯ cache misses
    - Mergesort? ↯ linear-size buffer needed (or slow)

What's Wrong with Quicksort?

think: `ints`

- When sorting **simple** objects
  1. **quicksort hard to beat**!
  2. memory transfers a substantial fraction of cost

- What if key **comparisons** are **expensive**?
  - ⇝ Quicksort is a factor > 1 off from optimal
    - Heapsort? ↯ cache misses
    - Mergesort? ↯ linear-size buffer needed (or slow)

What's Wrong with Quicksort?

think: `ints`

- When sorting **simple** objects
  1. **quicksort hard to beat**!
  2. memory transfers a substantial fraction of cost

- What if key **comparisons** are **expensive**?

  ⇝ Quicksort is a factor > 1 off from optimal

  - Heapsort? ↯ cache misses
  - Mergesort? ↯ linear-size buffer needed (or slow)

What's Wrong with Quicksort?

- When sorting **simple** objects (think: `ints`)
  1. **quicksort hard to beat**!
  2. memory transfers a substantial fraction of cost

- What if key **comparisons** are **expensive**?
  - ⇝ Quicksort is a factor > 1 off from optimal
    - Heapsort? ↯ cache misses
    - Mergesort? ↯ linear-size buffer needed (or slow)



Can quicksort help?

What is QuickXsort?

some sorting method

$$\text{Quick}\mathbf{X}\text{sort} = \text{Quicksort} + \mathbf{X}$$

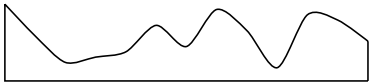
What is QuickXsort?

some sorting method

QuickXsort = Quicksort + X

QuickXsort(A[l..r])

1. Choose pivot P.
2. i = Partition(A[l..r], P)
   // subproblems: A[l..i − 1] and A[i + 1..r]
3. Sort one subproblem with X.
4. Sort the other subproblem with QuickXsort.



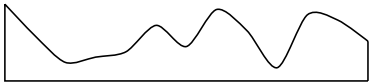
What is QuickXsort?

some sorting method

QuickXsort = Quicksort + X

QUICKXSORT(A[l..r])

1 Choose pivot P.
2 i = PARTITION(A[l..r], P)
// subproblems: A[l..i − 1] and A[i + 1..r]
3 Sort one subproblem with X.
4 Sort the other subproblem with QUICKXSORT.



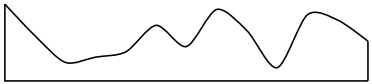
What is QuickXsort?

some sorting method

QuickXsort = Quicksort + X

QUICKXSORT(A[l..r])

1 Choose pivot P. (median of $k = 2t + 1$)
2 i = PARTITION(A[l..r], P)
// subproblems: A[l..i − 1] and A[i + 1..r]
3 Sort one subproblem with X.
4 Sort the other subproblem with QUICKXSORT.



What is QuickXsort?

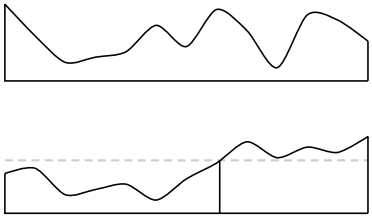
some sorting method

QuickXsort = Quicksort + X

QUICKXSORT(A[l..r])

median of $k = 2t + 1$

1 Choose pivot P.
2 i = PARTITION(A[l..r], P)
// subproblems: A[l..i − 1] and A[i + 1..r]
3 Sort one subproblem with X.
4 Sort the other subproblem with QUICKXSORT.



What is QuickXsort?

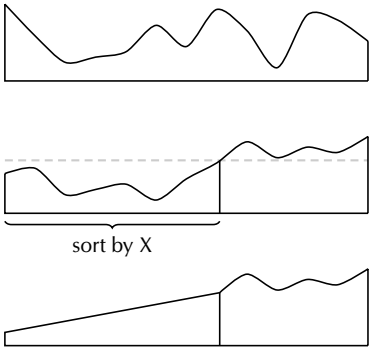
some sorting method

QuickXsort = Quicksort + X

QUICKXSORT(A[l..r])

median of $k = 2t + 1$

1. Choose pivot P.
2. i = PARTITION(A[l..r], P)
   // subproblems: A[l..i − 1] and A[i + 1..r]
3. Sort one subproblem with X.
4. Sort the other subproblem with QUICKXSORT.



What is QuickXsort?

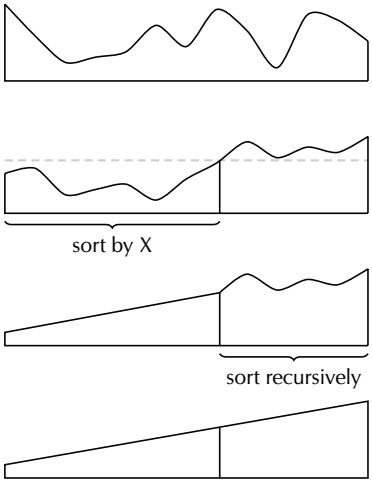
some sorting method

QuickXsort = Quicksort + X

QUICKXSORT(A[l..r])

median of k = 2t + 1

1. Choose pivot P.
2. i = PARTITION(A[l..r], P)
   // subproblems: A[l..i − 1] and A[i + 1..r]
3. Sort one subproblem with X.
4. Sort the other subproblem with QUICKXSORT.



What is QuickXsort?

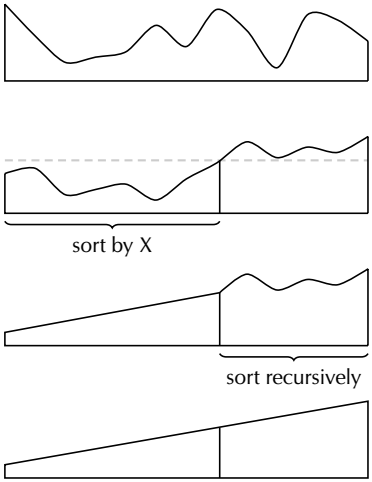
some sorting method

QuickXsort = Quicksort + X

QUICKXSORT(A[l..r])

median of $k = 2t + 1$

1. Choose pivot P.
2. i = PARTITION(A[l..r], P)
   // subproblems: A[l..i − 1] and A[i + 1..r]
3. **Sort one subproblem with X.**
4. Sort the other subproblem with QUICKXSORT.



What is QuickXsort?

some sorting method

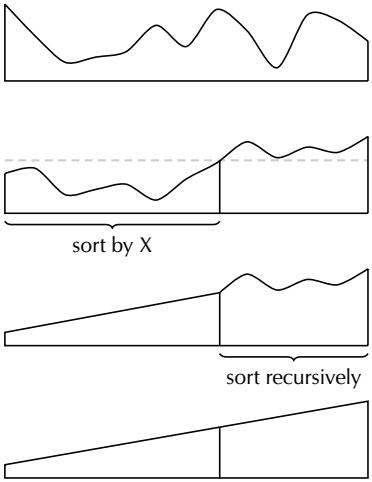
QuickXsort = Quicksort + X

QUICKXSORT(A[l..r])

median of k = 2t + 1

1 Choose pivot P.
2 i = PARTITION(A[l..r], P)
// subproblems: A[l..i − 1] and A[i + 1..r]
3 **Sort one subproblem with X.**
4 Sort the other subproblem with QUICKXSORT.

OK, but why? Why not simple use X?



What is QuickXsort?

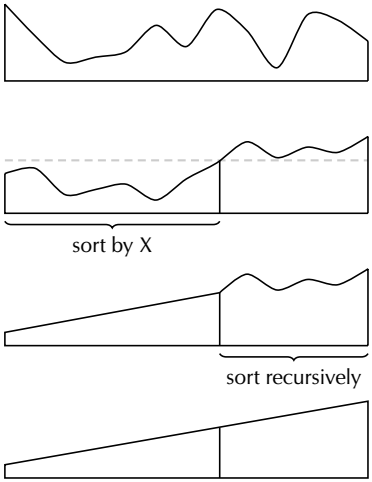
some sorting method

QuickXsort = Quicksort + X

QUICKXSORT(A[l..r])

1. Choose pivot P. (median of $k = 2t + 1$)
2. i = PARTITION(A[l..r], P)
   // subproblems: A[l..i − 1] and A[i + 1..r]
3. **Sort one subproblem with X,**
   using the other subproblem as buffer area.
4. Sort the other subproblem with QUICKXSORT.

OK, but why? Why not simple use X?



What is QuickXsort?

some sorting method

QuickXsort = Quicksort + X

QUICKXSORT(A[l..r])

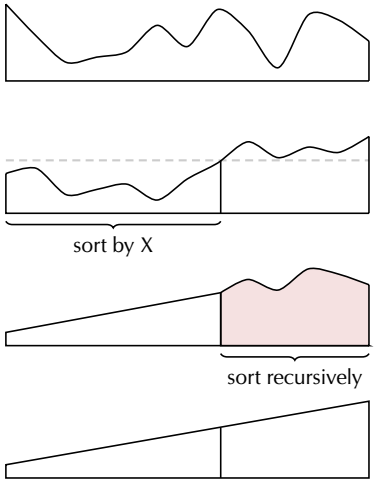
median of $k = 2t + 1$

1. Choose pivot P.
2. i = PARTITION(A[l..r], P)
   // subproblems: A[l..i − 1] and A[i + 1..r]
3. **Sort one subproblem with X,**
   using the other subproblem as buffer area.
4. Sort the other subproblem with QUICKXSORT.

⇝ *QuickXsort turns X into an **in-place** sorting method!*

- Here: **X = Mergesort**

⇝ inherit good comparison count, but work in-place!



What is QuickXsort?

some sorting method

QuickXsort = Quicksort + X

QUICKXSORT(A[l..r])

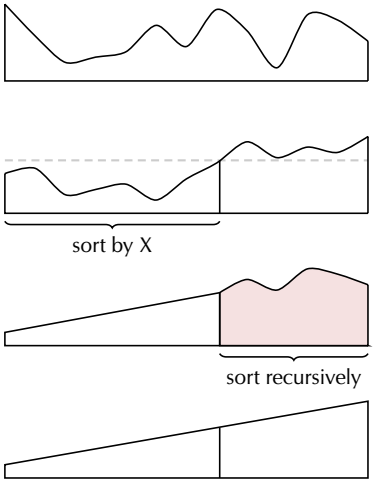
median of k = 2t + 1

1. Choose pivot P.
2. i = PARTITION(A[l..r], P)
   // subproblems: A[l..i − 1] and A[i + 1..r]
3. **Sort one subproblem with X,**
   using the other subproblem as buffer area.
4. Sort the other subproblem with QUICKXSORT.

⇝ *QuickXsort turns X into an **in-place** sorting method!*

- Here: **X = Mergesort**

⇝ inherit good comparison count, but work in-place!



What is QuickXsort?

some sorting method

QuickXsort = Quicksort + X

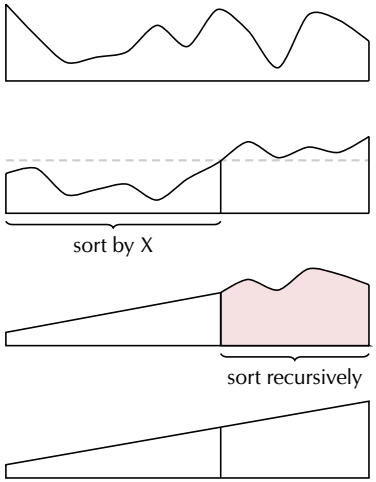
QUICKXSORT(A[l..r])

1 Choose pivot P. (median of $k = 2t + 1$)
2 i = PARTITION(A[l..r], P)
// subproblems: A[l..i − 1] and A[i + 1..r]
3 **Sort one subproblem with X,**
using the other subproblem as buffer area.
4 Sort the other subproblem with QUICKXSORT.

⇝ *QuickXsort turns X into an **in-place** sorting method!*

- Here: **X = Mergesort**

⇝ inherit good comparison count, but work in-place!



What is QuickXsort?

some sorting method

QuickXsort = Quicksort + X

QUICKXSORT(A[l..r])

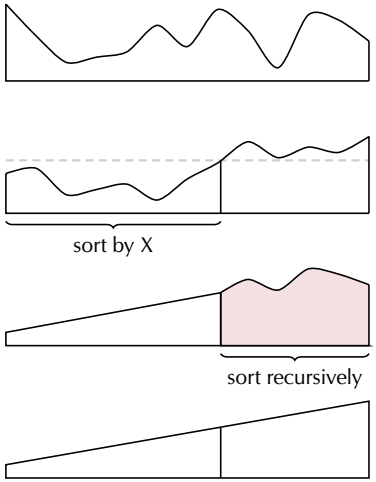
1. Choose pivot P. (median of $k = 2t + 1$)
2. i = PARTITION(A[l..r], P)
   // subproblems: A[l..i − 1] and A[i + 1..r]
3. **Sort one subproblem with X,**
   using the other subproblem as buffer area.
4. Sort the other subproblem with QUICKXSORT.

⇝ *QuickXsort turns X into an **in-place** sorting method!*

- Here: **X = Mergesort**

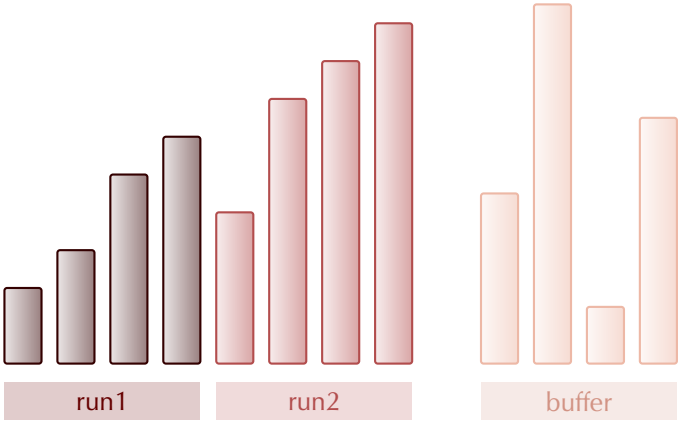
⇝ inherit good comparison count, but work in-place!

Or. . . do we?



Merging with swaps

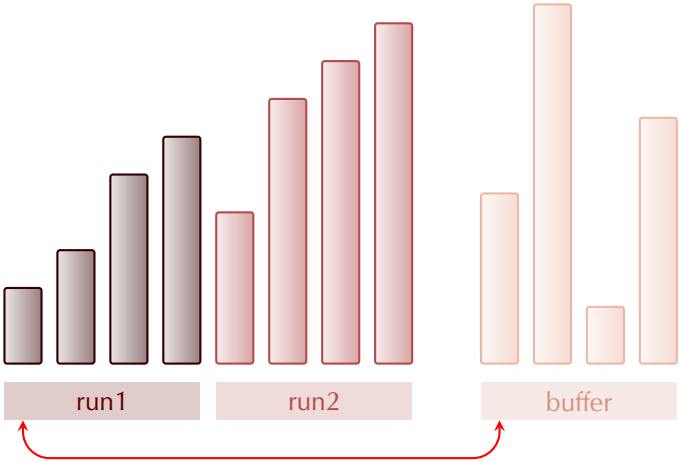
Important: *Buffer elements must stay intact.*



1. Swap run1 and buffer
2. Merge back (swap min)
3. runs merged,
   buffer restored (but permuted)

Merging with swaps

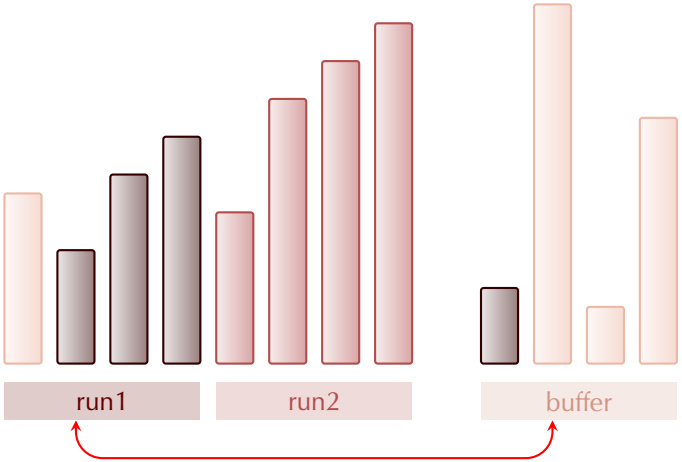
Important: *Buffer elements must stay intact.*



1. Swap run1 and buffer
2. Merge back (swap min)
3. runs merged,
   buffer restored (but permuted)

Merging with swaps

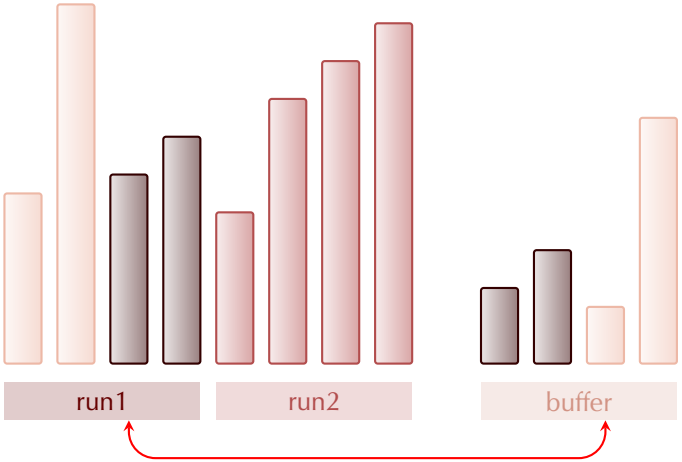
Important: *Buffer elements must stay intact.*



1. Swap run1 and buffer
2. Merge back (swap min)
3. runs merged,
   buffer restored (but permuted)

Merging with swaps

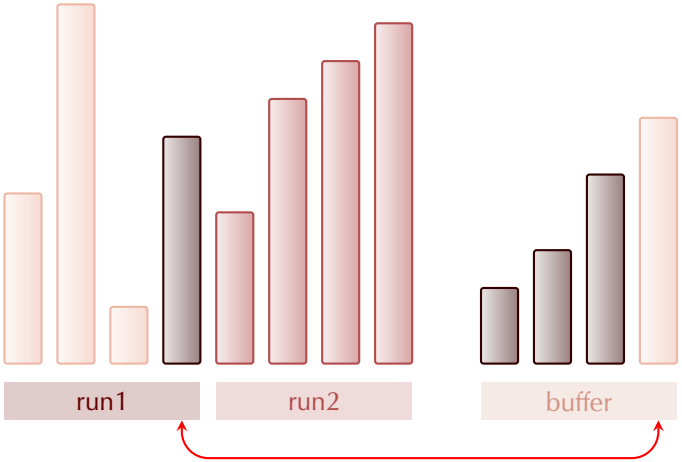
Important: *Buffer elements must stay intact.*



1. Swap run1 and buffer
2. Merge back (swap min)
3. runs merged,
   buffer restored (but permuted)

Merging with swaps

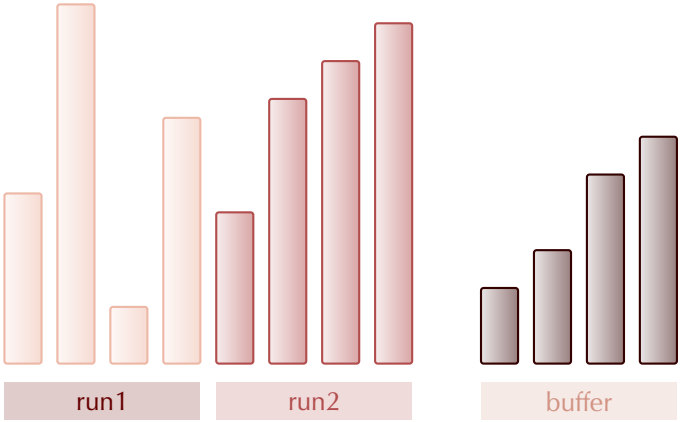
Important: *Buffer elements must stay intact.*



1. Swap run1 and buffer
2. Merge back (swap min)
3. runs merged,
   buffer restored (but permuted)

Merging with swaps

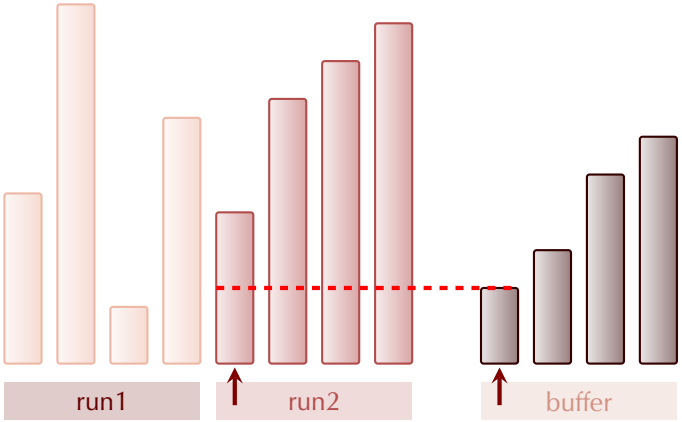
Important: *Buffer elements must stay intact.*



1. Swap run1 and buffer
2. Merge back (swap min)
3. runs merged,
   buffer restored (but permuted)

Merging with swaps

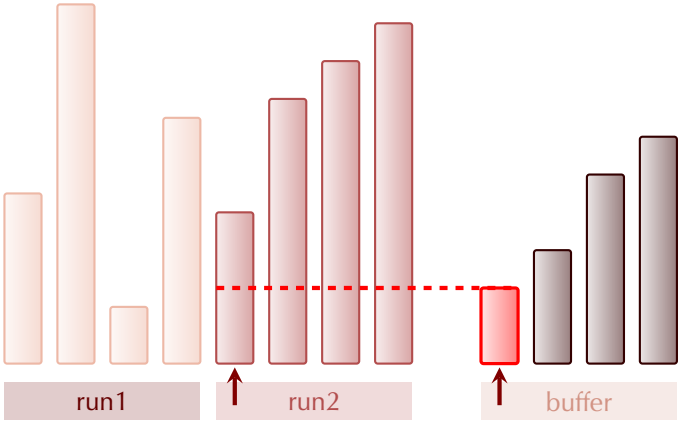
Important: *Buffer elements must stay intact.*



1. Swap run1 and buffer
2. Merge back (swap min)
3. runs merged,
   buffer restored (but permuted)

Merging with swaps

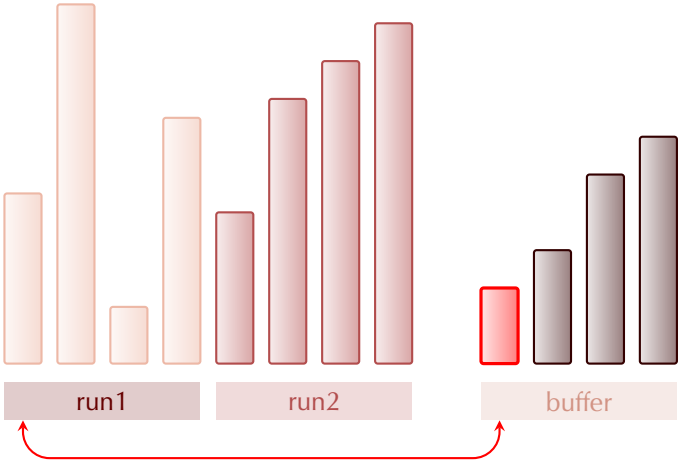
Important: *Buffer elements must stay intact.*



1. Swap run1 and buffer
2. Merge back (swap min)
3. runs merged,
   buffer restored (but permuted)

Merging with swaps

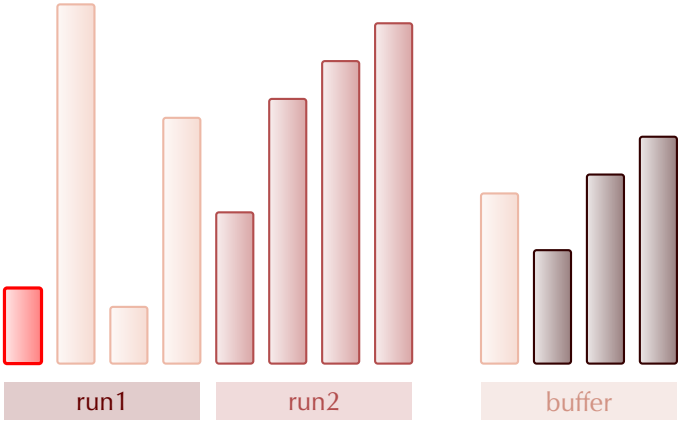
Important: *Buffer elements must stay intact.*



1. Swap run1 and buffer
2. Merge back (swap min)
3. runs merged,
   buffer restored (but permuted)

Merging with swaps

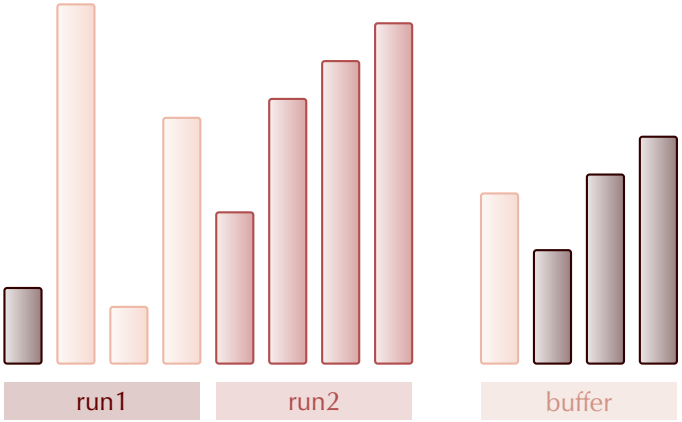
Important: *Buffer elements must stay intact.*



1. Swap run1 and buffer
2. Merge back (swap min)
3. runs merged,
   buffer restored (but permuted)

Merging with swaps

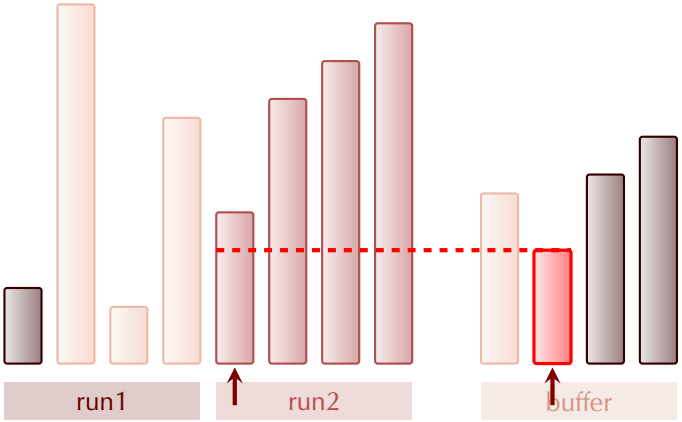
Important: *Buffer elements must stay intact.*



1. Swap run1 and buffer
2. Merge back (swap min)
3. runs merged,
   buffer restored (but permuted)

Merging with swaps

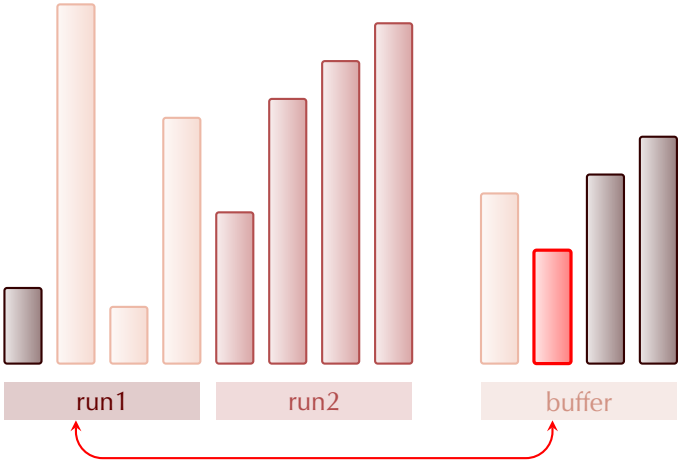
Important: *Buffer elements must stay intact.*



1. Swap run1 and buffer
2. Merge back (swap min)
3. runs merged,
   buffer restored (but permuted)

Merging with swaps

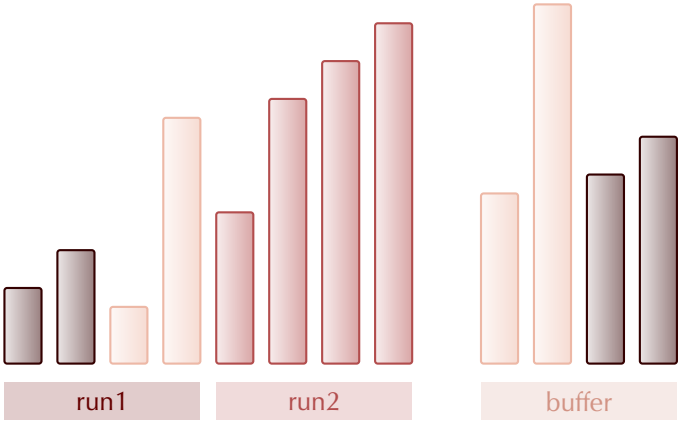
Important: *Buffer elements must stay intact.*



1. Swap run1 and buffer
2. Merge back (swap min)
3. runs merged,
   buffer restored (but permuted)

Merging with swaps

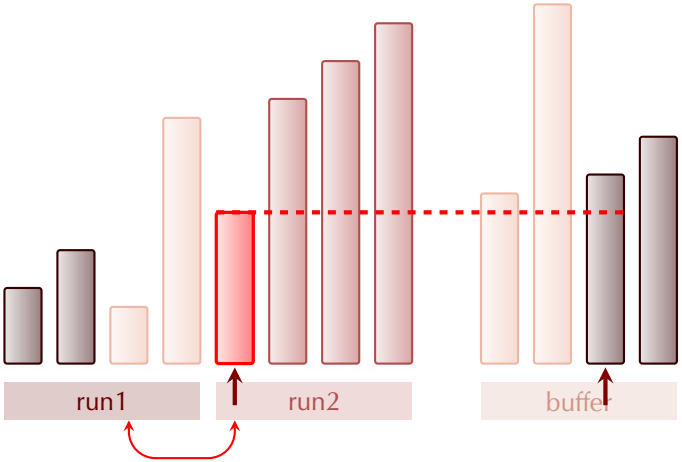
Important: *Buffer elements must stay intact.*



1. Swap run1 and buffer
2. Merge back (swap min)
3. runs merged,
   buffer restored (but permuted)

Merging with swaps

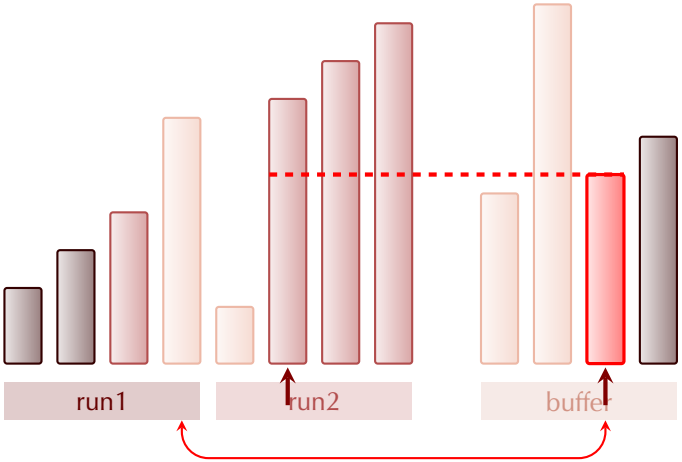
Important: *Buffer elements must stay intact.*



1. Swap run1 and buffer
2. Merge back (swap min)
3. runs merged,
   buffer restored (but permuted)

Merging with swaps

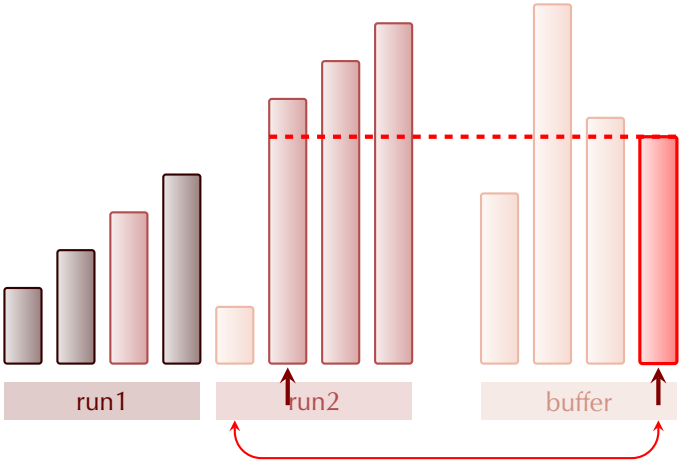
Important: *Buffer elements must stay intact.*



1. Swap run1 and buffer
2. Merge back (swap min)
3. runs merged,
   buffer restored (but permuted)

Merging with swaps

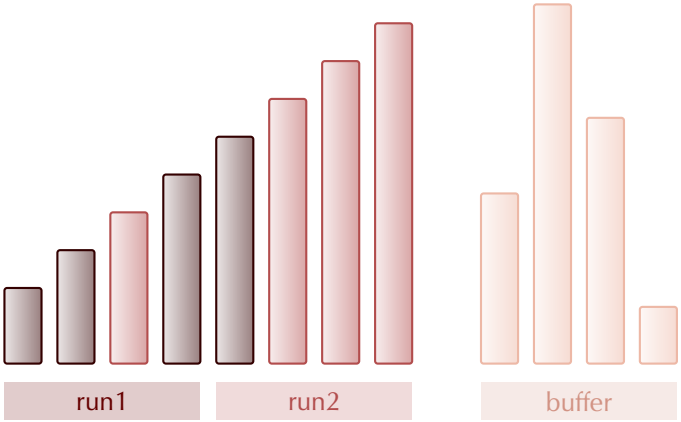
Important: *Buffer elements must stay intact.*



1. Swap run1 and buffer
2. Merge back (swap min)
3. runs merged,
   buffer restored (but permuted)

Merging with swaps

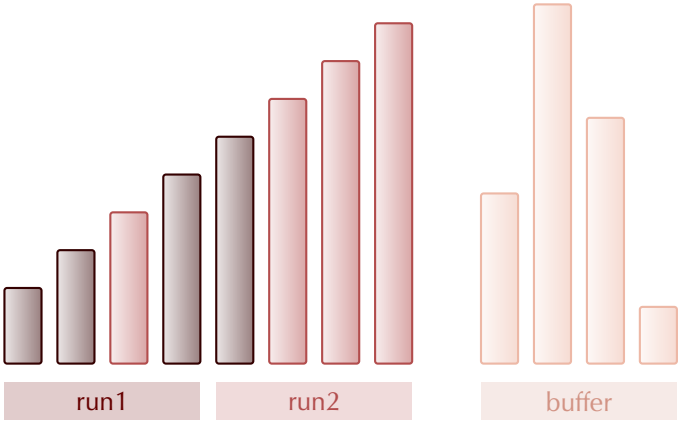
Important: *Buffer elements must stay intact.*



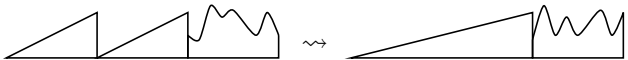
1. Swap run1 and buffer
2. Merge back (swap min)
3. runs merged,
   buffer restored (but permuted)

Merging with swaps

Important: *Buffer elements must stay intact.*



1. Swap run1 and buffer
2. Merge back (swap min)
3. runs merged,
   buffer restored (but permuted)



Anlysis of QuickXsort

Comparisons in QuickXsort

exact **average** case results
with pivots as **median** of $2t+1$ elements

$$c(n) \;=\; x(n) + q_t \cdot n \;\pm\; O(\log n)$$

Anlysis of QuickXsort

Comparisons in QuickXsort

exact **average** case results
with pivots as **median** of $2t+1$ elements

$$c(n) \;=\; x(n) \;+\; q_t \cdot n \;\pm\; O(\log n)$$

Anlysis of QuickXsort

Comparisons in QuickXsort

exact **average** case results
with pivots as **median** of $2t+1$ elements

same as X ...

$$c(n) \;=\; x(n) \;+\; q_t \cdot n \quad \pm \quad O(\log n)$$

Anlysis of QuickXsort

Comparisons in QuickXsort

exact **average** case results
with pivots as **median** of $2t+1$ elements

same as X …

$$c(n) \;=\; x(n) \;+\; q_t \cdot n \;\pm\; O(\log n)$$

… + "QuickXsort penalty"

Anlysis of QuickXsort

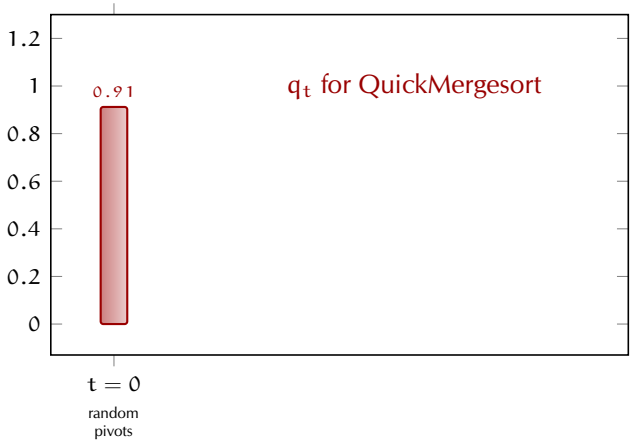
Comparisons in QuickXsort

exact **average** case results
with pivots as **median** of $2t+1$ elements

same as X …

$$c(n) = x(n) + q_t \cdot n \pm O(\log n)$$

… + "QuickXsort penalty"



Anlysis of QuickXsort

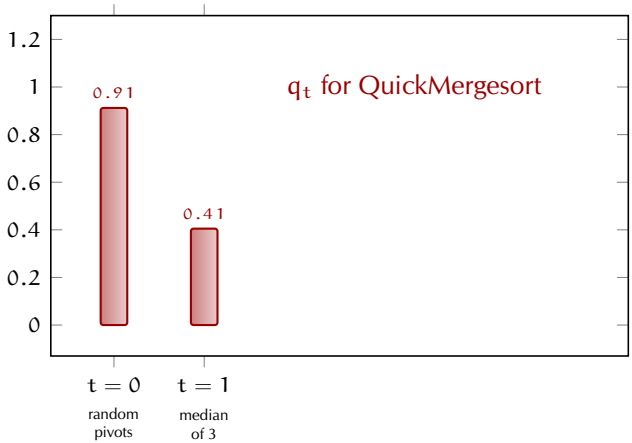
Comparisons in QuickXsort

exact **average** case results
with pivots as **median** of $2t+1$ elements

same as X …

$$c(n) = x(n) + q_t \cdot n \pm O(\log n)$$

… + "QuickXsort penalty"



Anlysis of QuickXsort

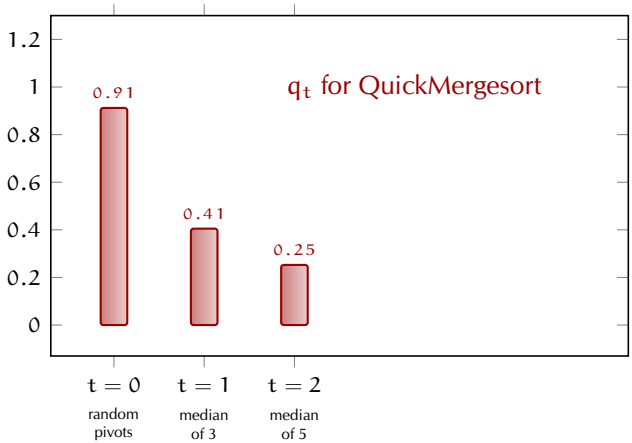
Comparisons in QuickXsort

exact **average** case results
with pivots as **median** of $2t+1$ elements

same as X ...

$$c(n) = x(n) + q_t \cdot n \pm O(\log n)$$

... + "QuickXsort penalty"



Anlysis of QuickXsort

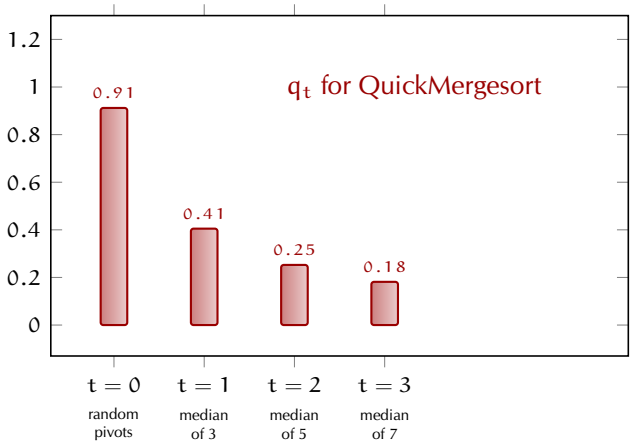
Comparisons in QuickXsort

exact **average** case results
with pivots as **median** of $2t+1$ elements

same as X …

$$c(n) = x(n) + q_t \cdot n \pm O(\log n)$$

… + "QuickXsort penalty"



Anlysis of QuickXsort

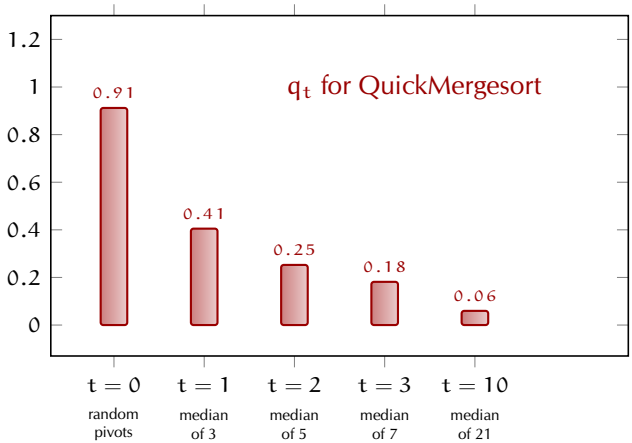
Comparisons in QuickXsort

exact **average** case results
with pivots as **median** of $2t+1$ elements

same as X …

$$c(n) = x(n) + q_t \cdot n \pm O(\log n)$$

… + "QuickXsort penalty"



Anlysis of QuickXsort

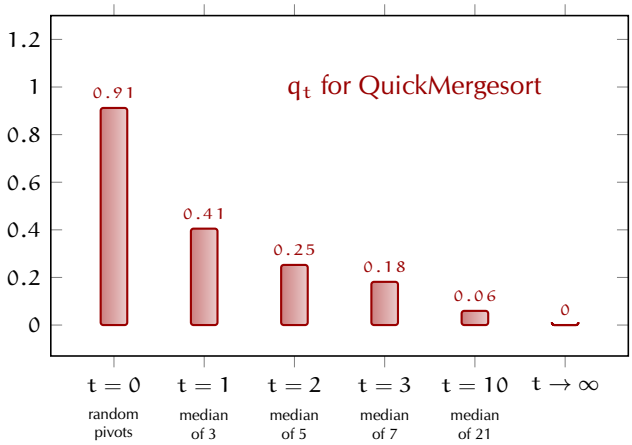
Comparisons in QuickXsort

exact **average** case results
with pivots as **median** of $2t+1$ elements

same as X …

$$c(n) = x(n) + q_t \cdot n \pm O(\log n)$$

… + "QuickXsort penalty"



Anlysis of QuickXsort

Comparisons in QuickXsort

exact **average** case results
with pivots as **median** of $2t+1$ elements

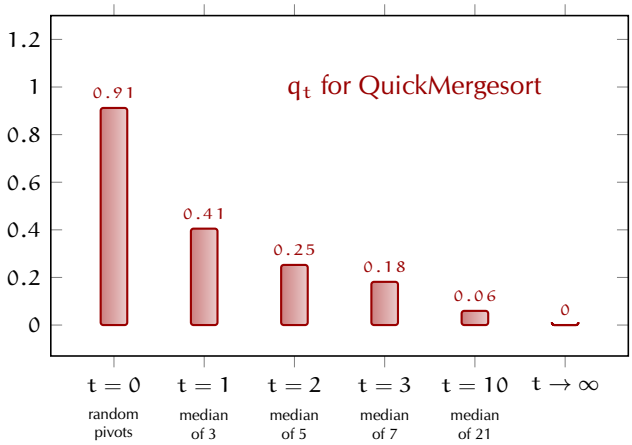
- Old quicksort wisdom rediscovered:
  small samples give most of gain
- **#comparisons** close to **optimal**
- running-time efficiency confirmed in experiments

⇝ *Can get **in-place** with tiny overhead*

same as X …

$$c(n) = x(n) + q_t \cdot n \pm O(\log n)$$

… + "QuickXsort penalty"



Anlysis of QuickXsort

Comparisons in QuickXsort

exact **average** case results
with pivots as **median** of $2t+1$ elements

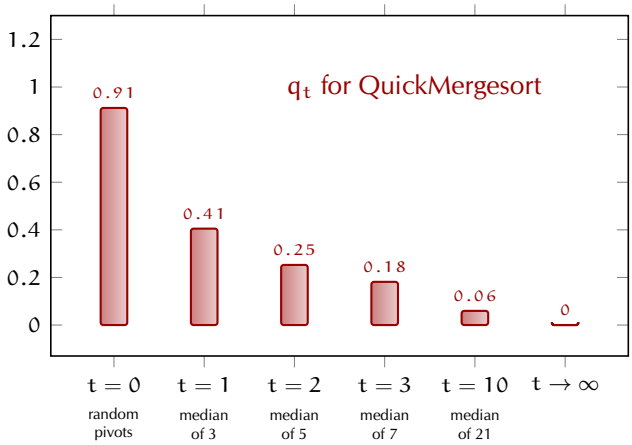
- Old quicksort wisdom rediscovered:
  small samples give most of gain
- **#comparisons** close to **optimal**
- running-time efficiency confirmed in experiments

⇝ *Can get **in-place** with tiny overhead*

same as X …

$$c(n) = x(n) + q_t \cdot n \pm O(\log n)$$

… + "QuickXsort penalty"



Anlysis of QuickXsort

Comparisons in QuickXsort

exact **average** case results
with pivots as **median** of $2t+1$ elements

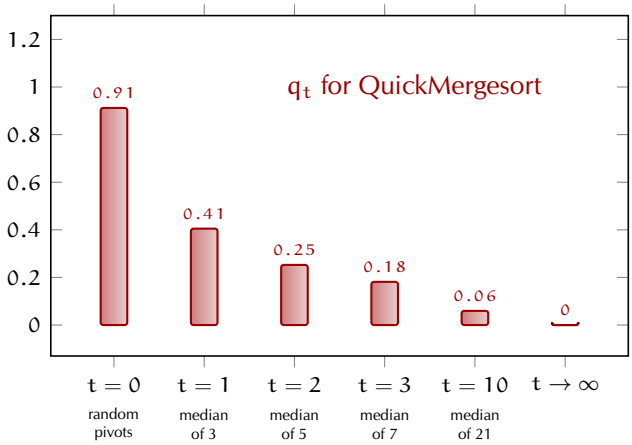
$$c(n) = x(n) + q_t \cdot n \pm O(\log n)$$

same as X …

… + "QuickXsort penalty"

- Old quicksort wisdom rediscovered: small samples give most of gain
- **#comparisons** close to **optimal**
- running-time efficiency confirmed in experiments

⇝ *Can get **in-place** with tiny overhead*



Anlysis of QuickXsort

Comparisons in QuickXsort

exact **average** case results
with pivots as **median** of $2t+1$ elements

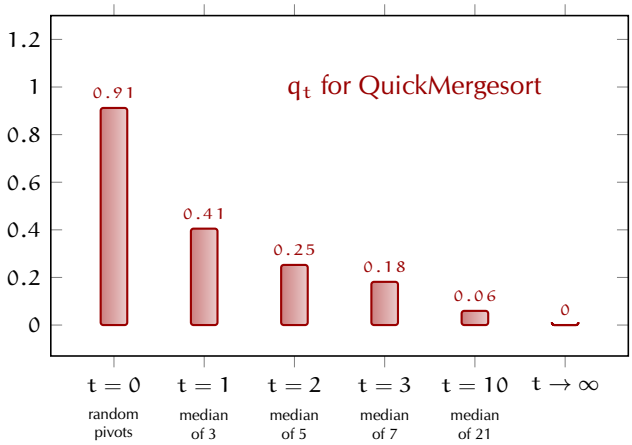
- Old quicksort wisdom rediscovered: small samples give most of gain
- **#comparisons** close to **optimal**
- running-time efficiency confirmed in experiments

⇝ *Can get **in-place** with tiny overhead*

same as X …

$$c(n) = x(n) + q_t \cdot n \pm O(\log n)$$

… + "QuickXsort penalty"



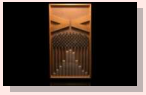
Outline



1 Context



2 Dual-Pivot Quicksort



3 Average-Case Analysis



4 Quicksort + X



5 Conclusion

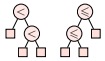


5 Conclusion

My decade with Quicksort

You have seen:

1. tricky **asymmetries** in *YBB Quicksort*
   ⇝ save a few comparisons



2. evidence for growing weight of memory transfer costs

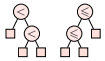
3. **Multiway Quicksort** reduces **memory transfer**
   - No other optimization of Quicksort achieves that

4. ***Quicksort continues to inspire*** new algorithms and algorithmists!

My decade with Quicksort

You have seen:

1. tricky **asymmetries** in *YBB Quicksort*
   ⇝ save a few comparisons



2. evidence for growing weight of memory transfer costs

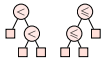
3. **Multiway Quicksort** reduces **memory transfer**
   - No other optimization of Quicksort achieves that

4. ***Quicksort continues to inspire*** new algorithms and algorithmists!

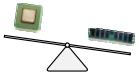
My decade with Quicksort

You have seen:

1. tricky **asymmetries** in *YBB Quicksort*
   ⇝ save a few comparisons



2. evidence for growing weight of memory transfer costs



3. **Multiway Quicksort** reduces **memory transfer**

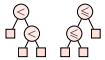
- No other optimization of Quicksort achieves that

4. ***Quicksort continues to inspire*** new algorithms and algorithmists!

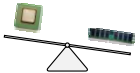
My decade with Quicksort

You have seen:

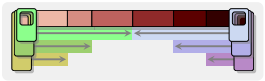
1. tricky **asymmetries** in *YBB Quicksort*
   ⇝ save a few comparisons



2. evidence for growing weight of memory transfer costs



3. **Multiway Quicksort** reduces **memory transfer**



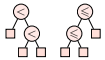
   - No other optimization of Quicksort achieves that

4. ***Quicksort continues to inspire*** new algorithms and algorithmists!

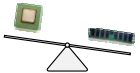
My decade with Quicksort

You have seen:

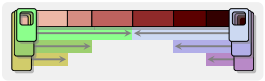
1. tricky **asymmetries** in *YBB Quicksort*
   ⇝ save a few comparisons



2. evidence for growing weight of memory transfer costs



3. **Multiway Quicksort** reduces **memory transfer**



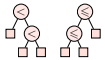
   - No other optimization of Quicksort achieves that

4. ***Quicksort continues to inspire*** new algorithms and algorithmists!

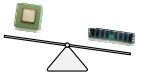
My decade with Quicksort

You have seen:

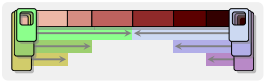
1. tricky **asymmetries** in *YBB Quicksort*
   ⇝ save a few comparisons



2. evidence for growing weight of memory transfer costs

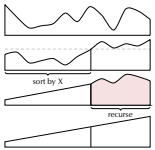


3. **Multiway Quicksort** reduces **memory transfer**



   - No other optimization of Quicksort achieves that

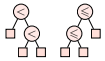
4. ***Quicksort continues to inspire*** new algorithms and algorithmists!



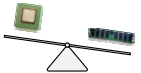
My decade with Quicksort

You have seen:

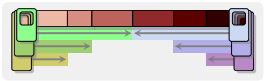
1. tricky **asymmetries** in *YBB Quicksort*
   ⇝ save a few comparisons



2. evidence for growing weight of memory transfer costs

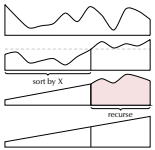


3. **Multiway Quicksort** reduces **memory transfer**



   - No other optimization of Quicksort achieves that

4. ***Quicksort continues to inspire*** new algorithms and algorithmists!



You have *not* seen:

- Treatment of **equal keys**

- Limiting **distributions** of costs

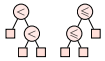
- **Sesquickselect** (cache-efficient selection)

- Intriguing interplay of parameters

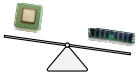
My decade with Quicksort

You have seen:

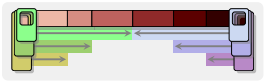
1. tricky **asymmetries** in *YBB Quicksort*
   ⇝ save a few comparisons



2. evidence for growing weight of memory transfer costs

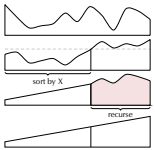


3. **Multiway Quicksort** reduces **memory transfer**



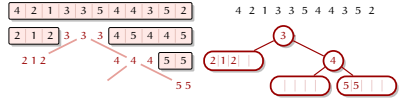
   - No other optimization of Quicksort achieves that

4. ***Quicksort continues to inspire*** new algorithms and algorithmists!



You have *not* seen:

- Treatment of **equal keys**



- Limiting **distributions** of costs

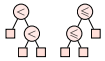
- **Sesquickselect** (cache-efficient selection)

- Intriguing interplay of parameters

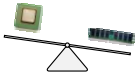
My decade with Quicksort

You have seen:

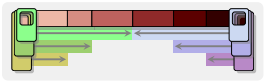
1. tricky **asymmetries** in *YBB Quicksort*
   ⇝ save a few comparisons



2. evidence for growing weight of memory transfer costs

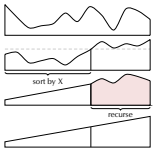


3. **Multiway Quicksort** reduces **memory transfer**



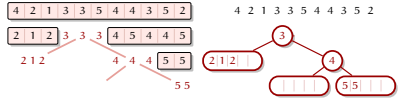
   - No other optimization of Quicksort achieves that

4. ***Quicksort continues to inspire*** new algorithms and algorithmists!

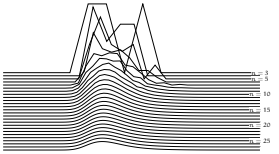


You have *not* seen:

- Treatment of **equal keys**



- Limiting **distributions** of costs



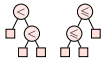
- **Sesquickselect** (cache-efficient selection)

- Intriguing interplay of parameters

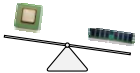
My decade with Quicksort

You have seen:

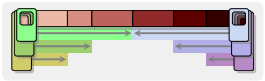
1. tricky **asymmetries** in *YBB Quicksort*
   ⇝ save a few comparisons



2. evidence for growing weight of memory transfer costs

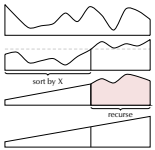


3. **Multiway Quicksort** reduces **memory transfer**



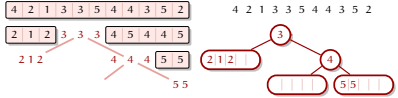
   - No other optimization of Quicksort achieves that

4. ***Quicksort continues to inspire*** new algorithms and algorithmists!

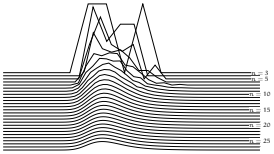


You have *not* seen:

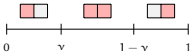
- Treatment of **equal keys**



- Limiting **distributions** of costs



- **Sesquickselect** (cache-efficient selection)

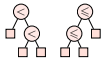


- Intriguing interplay of parameters

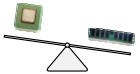
My decade with Quicksort

You have seen:

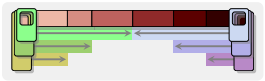
1. tricky **asymmetries** in *YBB Quicksort*
   ⇝ save a few comparisons



2. evidence for growing weight of memory transfer costs

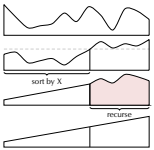


3. **Multiway Quicksort** reduces **memory transfer**



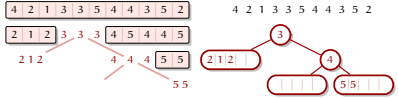
   - No other optimization of Quicksort achieves that

4. ***Quicksort continues to inspire*** new algorithms and algorithmists!

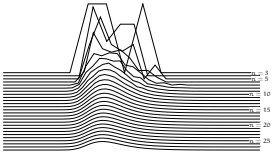


You have *not* seen:

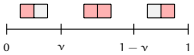
- Treatment of **equal keys**



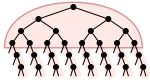
- Limiting **distributions** of costs



- **Sesquickselect** (cache-efficient selection)

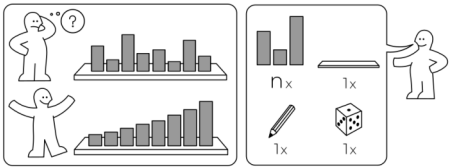


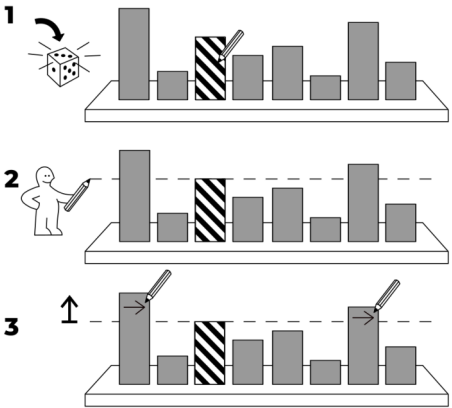
- Intriguing interplay of parameters

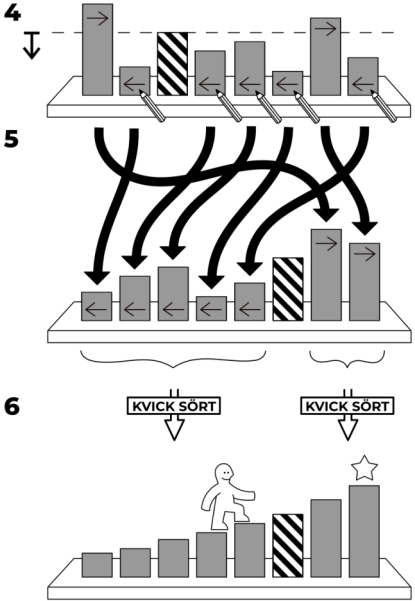


KVICK SÖRT

idea-instructions.com/quick-sort/
v1.2, CC by-nc-sa 4.0 IDEA

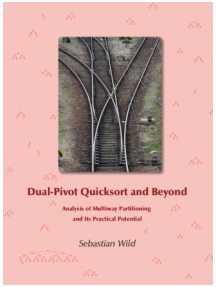








References



Most presented results appear in my thesis:

Dual-Pivot Quicksort And Beyond
Sebastian Wild • TU Kaiserslautern • 2016

Further Articles:

- ***Sesquickselect: One and a half pivots for cache-efficient selection***
  Conrado Martínez, Markus Nebel, Sebastian Wild • ANALCO • 2019
- ***QuickXsort – A Fast Sorting Scheme in Theory and Practice***
  Stefan Edelkamp, Armin Weiß, Sebastian Wild • Algorithmica • 2020
- ***Quicksort Is Optimal for Many Equal Keys***
  Sebastian Wild • ANALCO • 2018
- ***Analysis of Quickselect under Yaroslavskiy's Dual-Pivoting Algorithm***
  Sebastian Wild, Markus Nebel, Hosam Mahmoud • Algorithmica • 2016
- ***Analysis of Pivot Sampling in Dual-Pivot Quicksort***
  Markus Nebel, Sebastian Wild, Conrado Martínez • Algorithmica • 2016
- ***Analysis of Branch Misses in Quicksort***
  Conrado Martínez, Markus Nebel, Sebastian Wild • ANALCO • 2015
- ***Average Case and Distributional Analysis of Dual-Pivot Quicksort***
  Sebastian Wild, Markus Nebel, Ralph Neininger • ACM Trans. Alg. • 2015
- ***Average-Case Analysis of Java 7's Dual-Pivot Quicksort***
  Sebastian Wild, Markus Nebel • ESA • 2012

Basic Algorithms

```
ClassicQuicksort(A, left, right)
 1  if right ⩽ left then return
 2  P := A[right]
 3  k := left − 1;  g := right
 4  while true
 5        do k := k + 1 while A[k] < P end while
 6        do g := g − 1 while A[g] > P end while
 7        if k ⩾ g then break while end if
 8        Swap A[k] and A[g]
 9  end while
10  Swap A[k] and A[right]
11  ClassicQuicksort(A,  left  , k − 1)
12  ClassicQuicksort(A, k + 1,  right  )
```

Basic Algorithms

```
ClassicQuicksort(A, left, right)
 1  if right ⩽ left then return
 2  P := A[right]
 3  k := left − 1;  g := right
 4  while true
 5      do k := k + 1 while A[k] < P end while
 6      do g := g − 1 while A[g] > P end while
 7      if k ⩾ g then break while end if
 8      Swap A[k] and A[g]
 9  end while
10  Swap A[k] and A[right]
11  ClassicQuicksort(A, left , k − 1)
12  ClassicQuicksort(A, k + 1, right )
```

Basic Algorithms

```
ClassicQuicksort(A, left, right)
1  if right ⩽ left then return
2  P := A[right]
3  k := left − 1;  g := right
4  while true
5      do k := k + 1 while A[k] < P end while
6      do g := g − 1 while A[g] > P end while
7      if k ⩾ g then break while end if
8      Swap A[k] and A[g]
9  end while
10 Swap A[k] and A[right]
11 ClassicQuicksort(A,  left  , k − 1)
12 ClassicQuicksort(A, k + 1,  right )
```

- **Classic Quicksort**
  - *crossing-pointer* scheme
  - symmetric treatment

Basic Algorithms

```
ClassicQuicksort(A, left, right)
 1  if right ⩽ left then return
 2  P := A[right]
 3  k := left − 1;  g := right
 4  while true
 5        do k := k + 1 while A[k] < P end while
 6        do g := g − 1 while A[g] > P end while
 7        if k ⩾ g then break while end if
 8        Swap A[k] and A[g]
 9  end while
10  Swap A[k] and A[right]
11  ClassicQuicksort(A,  left  , k − 1)
12  ClassicQuicksort(A, k + 1,  right  )
```

- **Classic Quicksort**
  - *crossing-pointer* scheme 
  - symmetric treatment

Basic Algorithms

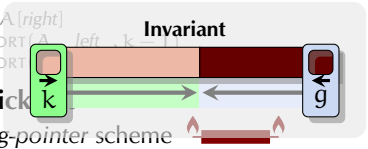
```
ClassicQuicksort(A, left, right)
 1  if right ⩽ left then return
 2  P := A[right]
 3  k := left − 1;  g := right
 4  while true
 5      do k := k + 1 while A[k] < P end while
 6      do g := g − 1 while A[g] > P end while
 7      if k ⩾ g then break while end if
 8      Swap A[k] and A[g]
 9  end while
10  Swap A[k] and A[right]
11  ClassicQuicksort(A,  left  , k − 1)
12  ClassicQuicksort(A, k + 1,  right  )
```

- **Classic Quicksort**
  - *crossing-pointer* scheme 
  - symmetric treatment 

Basic Algorithms

ClassicQuicksort(A, *left*, *right*)

```
1  if right ⩽ left then return
2  P := A[right]
3  k := left − 1;  g := right
4  while true
5      do k := k + 1 while A[k] < P end while
6      do g := g − 1 while A[g] > P end while
7      if k ⩾ g then break while end if
8      Swap A[k] and A[g]
9  end while
10 Swap A[k] and A[right]
11 ClassicQuicksort(A, left, k − 1)
12 ClassicQuicksort(A, k + 1, right)
```

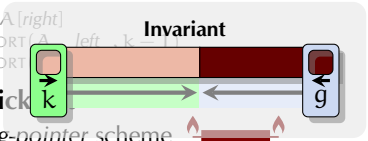


- **Classic Quicksort**
  - *crossing-pointer* scheme
  - symmetric treatment

Basic Algorithms

ClassicQuicksort(A, *left*, *right*)

```
1  if right ⩽ left then return
2  P := A[right]
3  k := left − 1;  g := right
4  while true
5      do k := k + 1 while A[k] < P end while
6      do g := g − 1 while A[g] > P end while
7      if k ⩾ g then break while end if
8      Swap A[k] and A[g]
9  end while
10 Swap A[k] and A[right]
11 ClassicQuicksort(A, left, k − 1)
12 ClassicQuicksort(A, k + 1, right)
```



- **Classic Quicksort**
  - *crossing-pointer* scheme
  - symmetric treatment
- **YBB Quicksort**
  - . . . looks messy

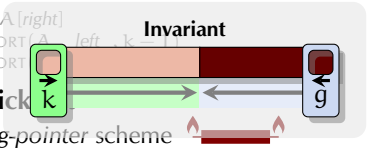
YbbQuicksort(A, *left*, *right*)

```
1  if right ⩽ left then return
2  P1 := A[left];   P2 := A[right]
3  if P1 > P2 then Swap P1 and P2 end if
4  ℓ := left + 1;   g := right − 1;   k := ℓ
5  while k ⩽ g
6      if A[k] < P1
7          Swap A[k] and A[ℓ] ;  ℓ := ℓ + 1
8      else if A[k] ⩾ P2
9          while A[g] > P2 and k < g
10             g := g − 1
11         end while
12         Swap A[k] and A[g] ;  g := g − 1
13         if A[k] < P1
14             Swap A[k] and A[ℓ] ;  ℓ := ℓ + 1
15         end if
16     end if
17     k := k + 1
18 end while
19 ℓ := ℓ − 1;   g := g + 1
20 Swap A[left] and A[ℓ] ; Swap A[right] and A[g]
21 YbbQuicksort(A,  left  , ℓ − 1 )
22 YbbQuicksort(A, ℓ + 1, g − 1)
23 YbbQuicksort(A, g + 1,  right  )
```

Basic Algorithms

ClassicQuicksort(A, *left*, *right*)

```
1  if right ⩽ left then return
2  P := A[right]
3  k := left − 1;  g := right
4  while true
5      do k := k + 1 while A[k] < P end while
6      do g := g − 1 while A[g] > P end while
7      if k ⩾ g then break while end if
8      Swap A[k] and A[g]
9  end while
10 Swap A[k] and A[right]
11 ClassicQuicksort(A, left, k − 1)
12 ClassicQuicksort(A, k + 1, right)
```



- **Classic Quicksort**
  - *crossing-pointer* scheme
  - symmetric treatment
- **YBB Quicksort**
  - . . . looks messy

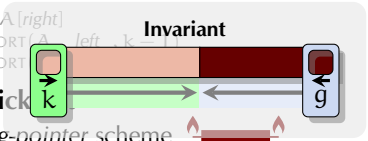
YbbQuicksort(A, *left*, *right*)

```
1  if right ⩽ left then return
2  P1 := A[left];   P2 := A[right]
3  if P1 > P2 then Swap P1 and P2 end if
4  ℓ := left + 1;   g := right − 1;   k := ℓ
5  while k ⩽ g
6      if A[k] < P1
7          Swap A[k] and A[ℓ] ;  ℓ := ℓ + 1
8      else if A[k] ⩾ P2
9          while A[g] > P2 and k < g
10             g := g − 1
11         end while
12         Swap A[k] and A[g] ;  g := g − 1
13         if A[k] < P1
14             Swap A[k] and A[ℓ] ;  ℓ := ℓ + 1
15         end if
16     end if
17     k := k + 1
18 end while
19 ℓ := ℓ − 1;   g := g + 1
20 Swap A[left] and A[ℓ] ; Swap A[right] and A[g]
21 YbbQuicksort(A, left , ℓ − 1)
22 YbbQuicksort(A, ℓ + 1, g − 1)
23 YbbQuicksort(A, g + 1, right )
```

Basic Algorithms

ClassicQuicksort$(A, left, right)$

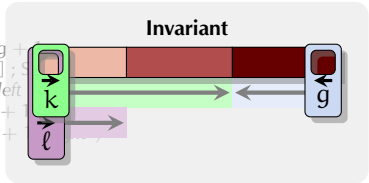
```
1  if right ⩽ left then return
2  P := A[right]
3  k := left − 1;  g := right
4  while true
5      do k := k + 1 while A[k] < P end while
6      do g := g − 1 while A[g] > P end while
7      if k ⩾ g then break while end if
8      Swap A[k] and A[g]
9  end while
10 Swap A[k] and A[right]
11 ClassicQuicksort(A, left, k − 1)
12 ClassicQuicksort(A, k + 1, right)
```



- **Classic Quicksort**
  - *crossing-pointer* scheme
  - symmetric treatment
- **YBB Quicksort**
  - . . . looks messy

YbbQuicksort$(A, left, right)$

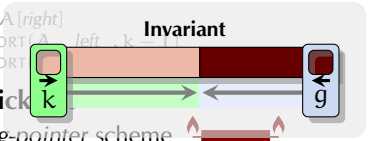
```
1  if right ⩽ left then return
2  P1 := A[left];   P2 := A[right]
3  if P1 > P2 then Swap P1 and P2 end if
4  ℓ := left + 1;   g := right − 1;   k := ℓ
5  while k ⩽ g
6      if A[k] < P1
7          Swap A[k] and A[ℓ] ;  ℓ := ℓ + 1
8      else if A[k] ⩾ P2
9          while A[g] > P2 and k < g
10             g := g − 1
11         end while
12         Swap A[k] and A[g] ;  g := g − 1
13         if A[k] < P1
14             Swap A[k] and A[ℓ] ;  ℓ := ℓ + 1
15         end if
16     end if
17     k := k + 1
18 end while
19 ℓ := ℓ − 1;   g := g
20 Swap A[left] and A[ℓ] ;
21 YbbQuicksort(A, left
22 YbbQuicksort(A, ℓ +
23 YbbQuicksort(A, g +
```



Basic Algorithms

ClassicQuicksort$(A, \mathit{left}, \mathit{right})$

```
1  if right ⩽ left then return
2  P := A[right]
3  k := left − 1;  g := right
4  while true
5      do k := k + 1 while A[k] < P end while
6      do g := g − 1 while A[g] > P end while
7      if k ⩾ g then break while end if
8      Swap A[k] and A[g]
9  end while
10 Swap A[k] and A[right]
11 ClassicQuicksort(A, left, k − 1)
12 ClassicQuicksort(A, k + 1, right)
```



- **Classic Quicksort**
  - *crossing-pointer* scheme
  - symmetric treatment

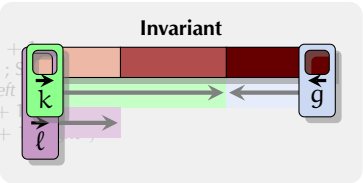


- **YBB Quicksort**
  - . . . looks messy



YbbQuicksort$(A, \mathit{left}, \mathit{right})$

```
1  if right ⩽ left then return
2  P1 := A[left];   P2 := A[right]
3  if P1 > P2 then Swap P1 and P2 end if
4  ℓ := left + 1;   g := right − 1;   k := ℓ
5  while k ⩽ g
6      if A[k] < P1
7          Swap A[k] and A[ℓ] ;  ℓ := ℓ + 1
8      else if A[k] ⩾ P2
9          while A[g] > P2 and k < g
10             g := g − 1
11         end while
12         Swap A[k] and A[g] ;  g := g − 1
13         if A[k] < P1
14             Swap A[k] and A[ℓ] ;  ℓ := ℓ + 1
15         end if
16     end if
17     k := k + 1
18 end while
19 ℓ := ℓ − 1;   g := g
20 Swap A[left] and A[ℓ] ;
21 YbbQuicksort(A, left
22 YbbQuicksort(A, ℓ +
23 YbbQuicksort(A, g +
```

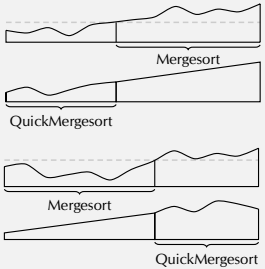


QuickMergesort

QUICKMERGESORT($A[l..r]$)

```
1  Choose pivot P.
2  i = PARTITION(A[l..r], P)

5  if ...
6      MERGESORT(A[i+1..r]) using buffer A[l..i-1]
7      QUICKMERGESORT(A[l..i-1]).
8  else
9      MERGESORT(A[l..i-1]) using buffer A[i+1..r]
10     QUICKMERGESORT(A[i..r]).
11 end if
```

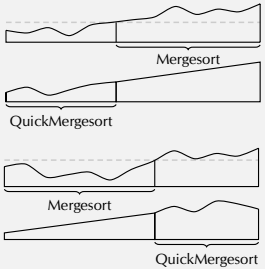


QuickMergesort

QUICKMERGESORT($A[l..r]$)

1 Choose pivot P.
2 $i = \text{PARTITION}(A[l..r], P)$
3 $J_1 := i - l; \quad J_2 := r - i$

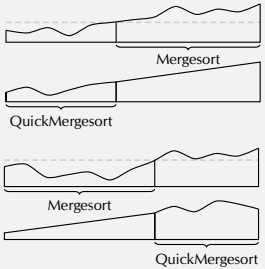
5 **if** ...
6 MERGESORT($A[i+1..r]$) using buffer $A[l..i-1]$
7 QUICKMERGESORT($A[l..i-1]$).
8 **else**
9 MERGESORT($A[l..i-1]$) using buffer $A[i+1..r]$
10 QUICKMERGESORT($A[i..r]$).
11 **end if**



QuickMergesort

QUICKMERGESORT($A[l..r]$)

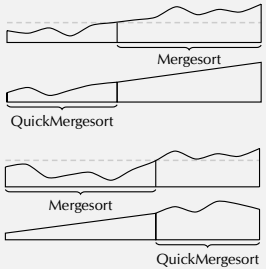
1 Choose pivot P.
2 i = PARTITION($A[l..r]$, P)
3 $J_1 := i - l$; $J_2 := r - i$
4 *largerWithX* := $\min\{J_1, J_2\} \geqslant \lfloor \frac{1}{2} \max\{J_1, J_2\} \rfloor$
5 **if** ...
6 MERGESORT($A[i+1..r]$) using buffer $A[l..i-1]$
7 QUICKMERGESORT($A[l..i-1]$).
8 **else**
9 MERGESORT($A[l..i-1]$) using buffer $A[i+1..r]$
10 QUICKMERGESORT($A[i..r]$).
11 **end if**



QuickMergesort

QUICKMERGESORT($A[l..r]$)

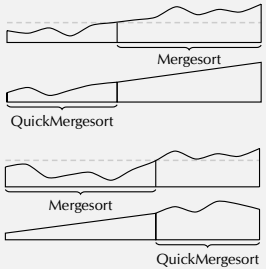
1 Choose pivot P.
2 i = PARTITION($A[l..r]$, P)
3 $J_1 := i - l$; $J_2 := r - i$
4 *largerWithX* $:= \min\{J_1, J_2\} \geqslant \left\lfloor \frac{1}{2} \max\{J_1, J_2\} \right\rfloor$
5 **if** $(\textit{largerWithX} \wedge J_2 \geqslant J_1)$
6 MERGESORT($A[i+1..r]$) using buffer $A[l..i-1]$
7 QUICKMERGESORT($A[l..i-1]$).
8 **else**
9 MERGESORT($A[l..i-1]$) using buffer $A[i+1..r]$
10 QUICKMERGESORT($A[i..r]$).
11 **end if**



QuickMergesort

QUICKMERGESORT($A[l..r]$)

1 Choose pivot P.
2 $i = \text{PARTITION}(A[l..r], P)$
3 $J_1 := i - l$; $J_2 := r - i$
4 *largerWithX* $:= \min\{J_1, J_2\} \geqslant \left\lfloor \frac{1}{2} \max\{J_1, J_2\} \right\rfloor$
5 **if** $(\textit{largerWithX} \wedge J_2 \geqslant J_1) \ \vee \ (\neg \textit{largerWithX} \wedge J_2 < J_1)$
6 MERGESORT($A[i+1..r]$) using buffer $A[l..i-1]$
7 QUICKMERGESORT($A[l..i-1]$).
8 **else**
9 MERGESORT($A[l..i-1]$) using buffer $A[i+1..r]$
10 QUICKMERGESORT($A[i..r]$).
11 **end if**



What happened? – Details

What do comparisons, write accesses, Bytecodes and branch misses have in common?

$\rightsquigarrow$ they happen **inside** the **CPU**

What happened? – Details

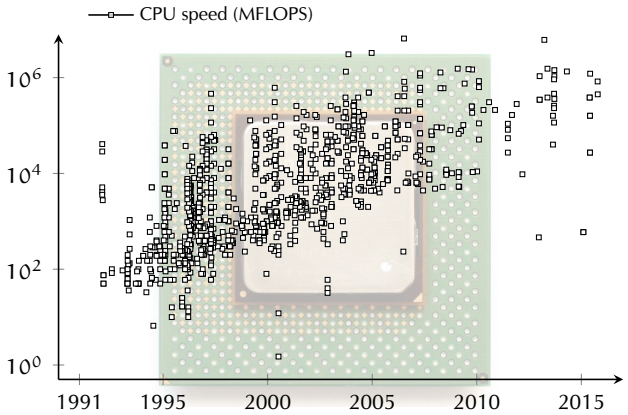
What do comparisons, write accesses, Bytecodes and branch misses have in common?
$\rightsquigarrow$ they happen **inside** the **CPU**

What happened? – Details

What do comparisons, write accesses, Bytecodes and branch misses have in common?
$\rightsquigarrow$ they happen **inside** the **CPU** ...but we need to get the data there, as well!

What happened? – Details

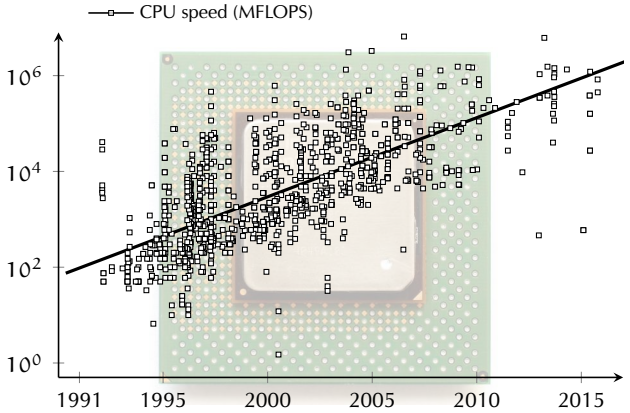
What do comparisons, write accesses, Bytecodes and branch misses have in common?
$\rightsquigarrow$ they happen **inside** the **CPU** …but we need to get the data there, as well!



STREAM Benchmark data with linear regression
`www.cs.virginia.edu/stream/by_date/Balance.html`

What happened? – Details

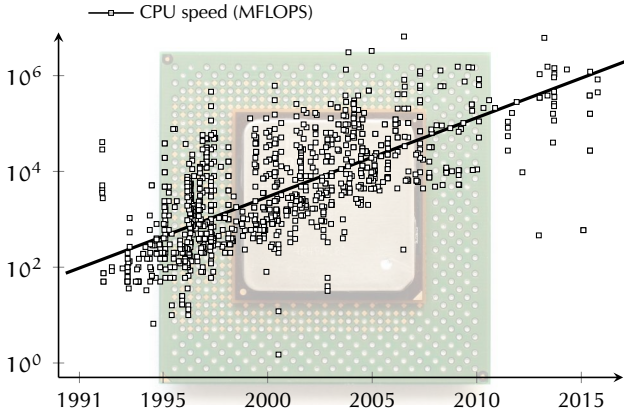
What do comparisons, write accesses, Bytecodes and branch misses have in common?
$\rightsquigarrow$ they happen **inside** the **CPU** …but we need to get the data there, as well!



STREAM Benchmark data with linear regression
www.cs.virginia.edu/stream/by_date/Balance.html

What happened? – Details

What do comparisons, write accesses, Bytecodes and branch misses have in common?
$\rightsquigarrow$ they happen **inside** the **CPU** ...but we need to get the data there, as well!

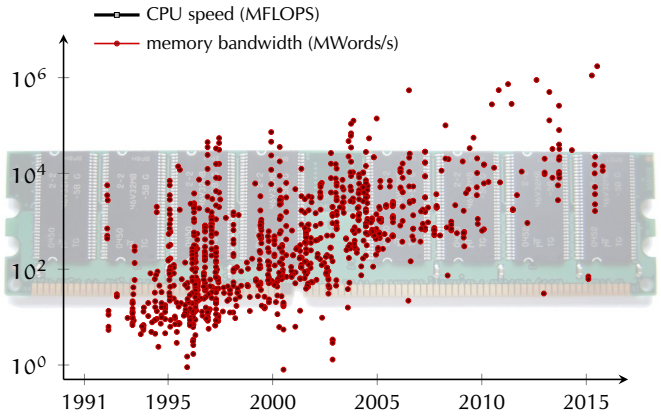


Average annual growth rate:
CPU speed: 46%

STREAM Benchmark data with linear regression
www.cs.virginia.edu/stream/by_date/Balance.html

What happened? – Details

What do comparisons, write accesses, Bytecodes and branch misses have in common?
$\rightsquigarrow$ they happen **inside** the **CPU** …but we need to get the data there, as well!



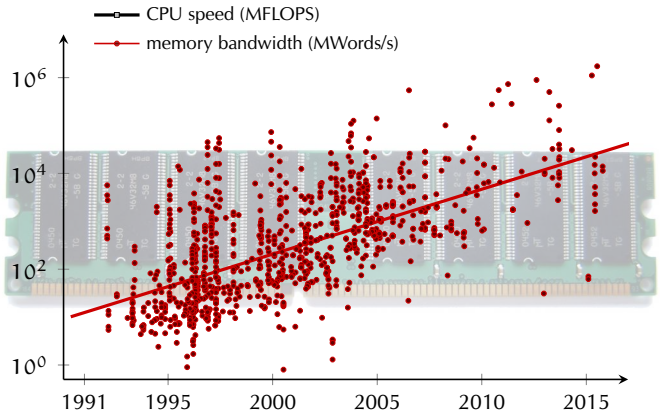
Average annual growth rate:

CPU speed:	46%

STREAM Benchmark data with linear regression
`www.cs.virginia.edu/stream/by_date/Balance.html`

What happened? – Details

What do comparisons, write accesses, Bytecodes and branch misses have in common?
$\rightsquigarrow$ they happen **inside** the **CPU** …but we need to get the data there, as well!



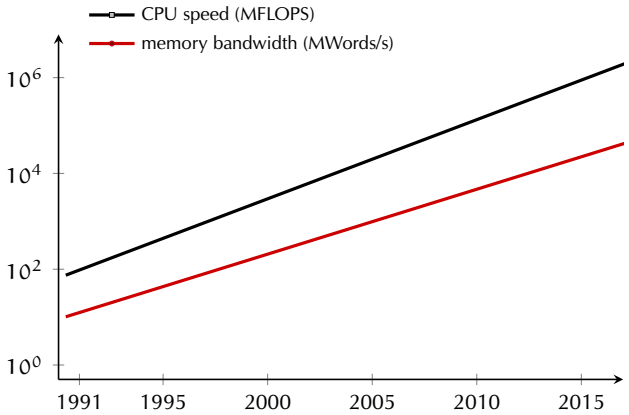
Average annual growth rate:

CPU speed: 46%
Memory Bandwidth: 37%

STREAM Benchmark data with linear regression
`www.cs.virginia.edu/stream/by_date/Balance.html`

What happened? – Details

What do comparisons, write accesses, Bytecodes and branch misses have in common?
$\rightsquigarrow$ they happen **inside** the **CPU** ...but we need to get the data there, as well!



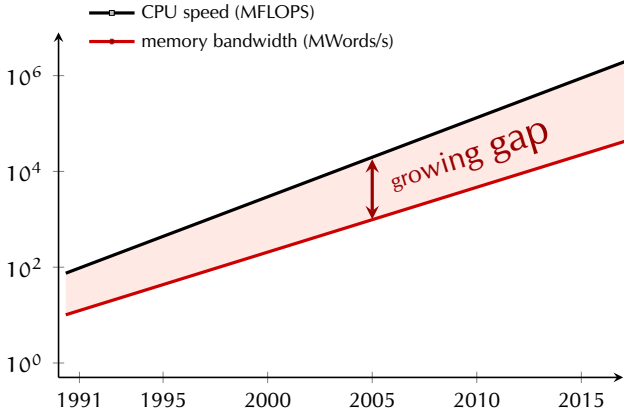
Average annual growth rate:

CPU speed: 46%
Memory Bandwidth: 37%

STREAM Benchmark data with linear regression
`www.cs.virginia.edu/stream/by_date/Balance.html`

What happened? – Details

What do comparisons, write accesses, Bytecodes and branch misses have in common?
⇝ they happen **inside** the **CPU** …but we need to get the data there, as well!



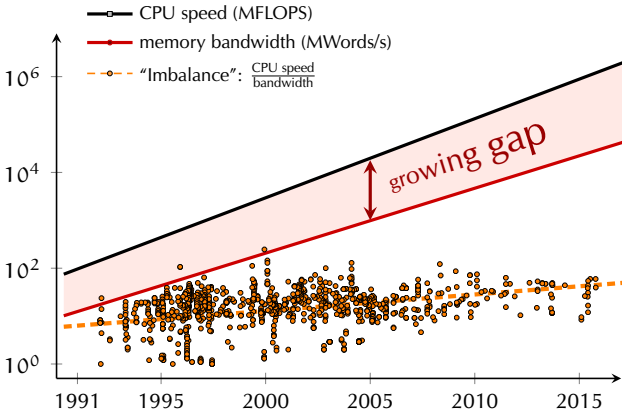
Average annual growth rate:

CPU speed: 46%
Memory Bandwidth: 37%

STREAM Benchmark data with linear regression
`www.cs.virginia.edu/stream/by_date/Balance.html`

What happened? – Details

What do comparisons, write accesses, Bytecodes and branch misses have in common?
$\rightsquigarrow$ they happen **inside** the **CPU** …but we need to get the data there, as well!



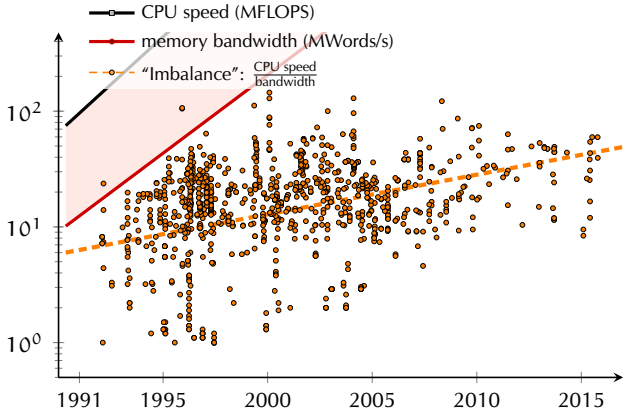
Average annual growth rate:

CPU speed: 46%
Memory Bandwidth: 37%

STREAM Benchmark data with linear regression
www.cs.virginia.edu/stream/by_date/Balance.html

What happened? – Details

What do comparisons, write accesses, Bytecodes and branch misses have in common?
⇝ they happen **inside** the **CPU** …but we need to get the data there, as well!



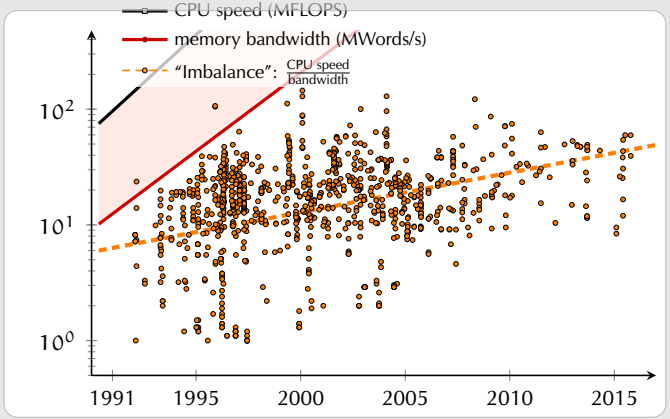
STREAM Benchmark data with linear regression
www.cs.virginia.edu/stream/by_date/Balance.html

Average annual growth rate:

CPU speed:	46%
Memory Bandwidth:	37%
⇝ Imbalance:	5%

What happened? – Details

What do comparisons, write accesses, Bytecodes and branch misses have in common?
⇝ they happen **inside** the **CPU** …but we need to get the data there, as well!



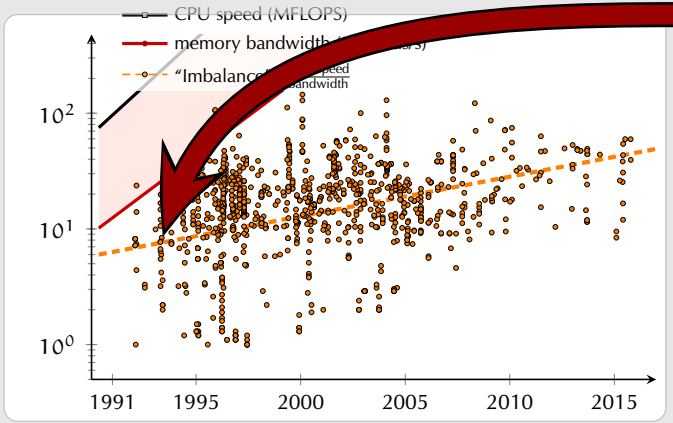
Average annual growth rate:

CPU speed:	46%
Memory Bandwidth:	37%
⇝ Imbalance:	5%

STREAM Benchmark data with linear regression
www.cs.virginia.edu/stream/by_date/Balance.html

What happened? – Details

What do comparisons, write accesses, Bytecodes and br
⇝ they happen **inside** the **CPU** …but we need to



Classic Quicksort (Bentley & McIlroy) **1993**

- 1 mem transfer ≈ 7 instructions

⇝ focus on CPU costs

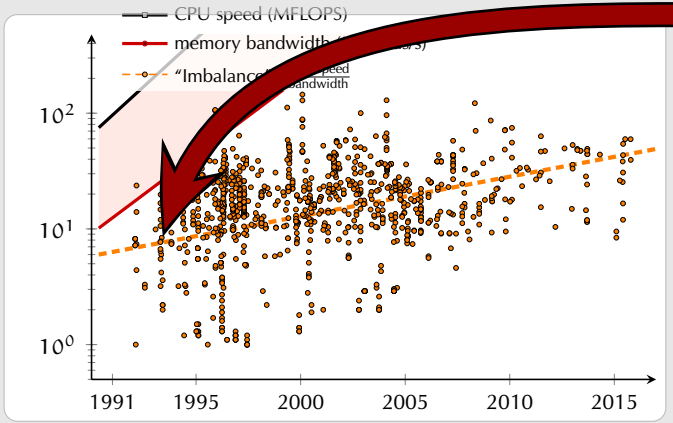
Average annual growth rate:

CPU speed: 46%
Memory Bandwidth: 37%
⇝ Imbalance: 5%

STREAM Benchmark data with linear regression
www.cs.virginia.edu/stream/by_date/Balance.html

What happened? – Details

What do comparisons, write accesses, Bytecodes and br
⇝ they happen **inside** the **CPU** …but we need to



Classic Quicksort (Bentley & McIlroy) **1993**

- 1 mem transfer ≈ 7 instructions

⇝ focus on CPU costs

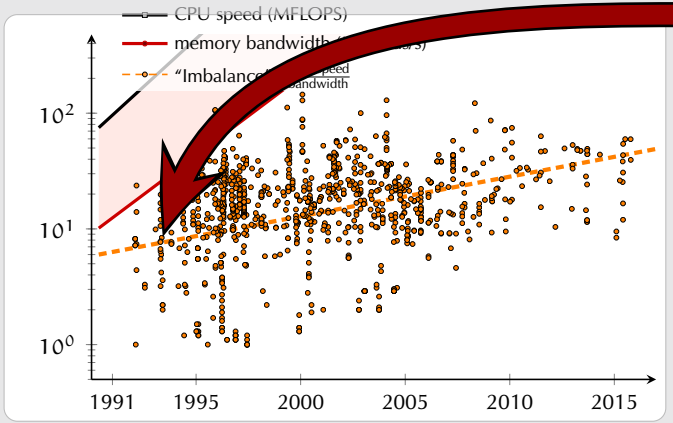
Average annual growth rate:

CPU speed: 46%
Memory Bandwidth: 37%
⇝ Imbalance: 5%

STREAM Benchmark data with linear regression
www.cs.virginia.edu/stream/by_date/Balance.html

What happened? – Details

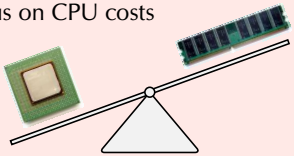
What do comparisons, write accesses, Bytecodes and br
⇝ they happen **inside** the **CPU** …but we need to



Classic Quicksort (Bentley & McIlroy) **1993**

- 1 mem transfer ≈ 7 instructions

⇝ focus on CPU costs



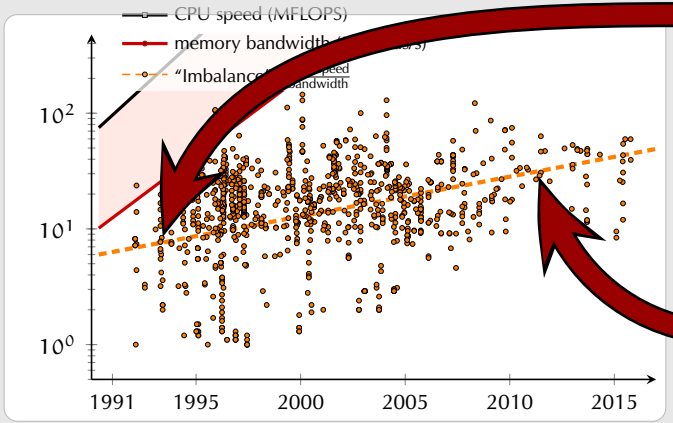
Average annual growth rate:

CPU speed: 46%
Memory Bandwidth: 37%
⇝ Imbalance: 5%

STREAM Benchmark data with linear regression
www.cs.virginia.edu/stream/by_date/Balance.html

What happened? – Details

What do comparisons, write accesses, Bytecodes and br
⇝ they happen **inside** the **CPU** …but we need to

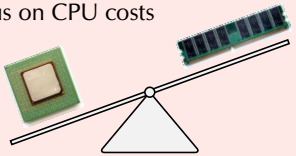


STREAM Benchmark data with linear regression
www.cs.virginia.edu/stream/by_date/Balance.html

Classic Quicksort (Bentley & McIlroy) **1993**

- 1 mem transfer ≈ 7 instructions

⇝ focus on CPU costs



Average annual growth rate:

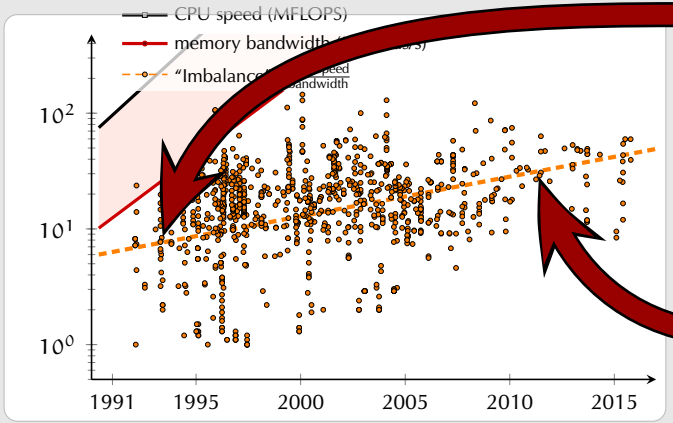
YBB Quicksort, **2009 – today**

- 1 transfer ≈ 30 instructions

⇝ focus on CPU and bandwidth

What happened? – Details

What do comparisons, write accesses, Bytecodes and br
⇝ they happen **inside** the **CPU** …but we need to

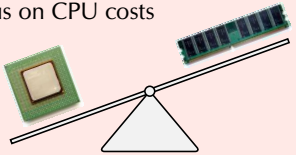


STREAM Benchmark data with linear regression
www.cs.virginia.edu/stream/by_date/Balance.html

Classic Quicksort (Bentley & McIlroy) **1993**

- 1 mem transfer ≈ 7 instructions

⇝ focus on CPU costs



Average annual growth rate:

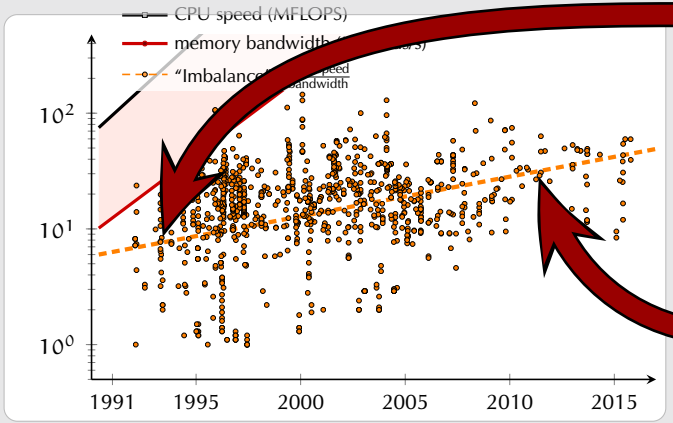
YBB Quicksort, **2009 – today**

- 1 transfer ≈ 30 instructions

⇝ focus on CPU and bandwidth

What happened? – Details

What do comparisons, write accesses, Bytecodes and br
⇝ they happen **inside** the **CPU** …but we need to

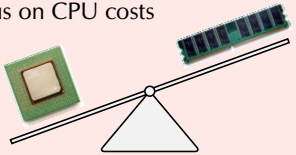


STREAM Benchmark data with linear regression
www.cs.virginia.edu/stream/by_date/Balance.html

Classic Quicksort (Bentley & McIlroy) **1993**

- 1 mem transfer ≈ 7 instructions

⇝ focus on CPU costs

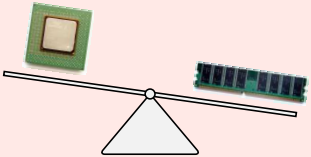


Average annual growth rate:

YBB Quicksort, **2009 – today**

- 1 transfer ≈ 30 instructions

⇝ focus on CPU **and bandwidth**



Icons made by *Freepik*, *Gregor Cresnar*, *Those Icons*, *Smashicons*, *Good Ware*, *Pause08*, and *Madebyoliver* from `www.flaticon.com`.

Vector graphics from Pressfoto, brgfx, macrovector and Jannoon028 on freepik.com
Top Trumps Card from `winningmoves.co.uk`.
3-way railway switch by *Asiano*
processor by *Konstantin Lanzet*
RAM module by *Utente:Sassospicco*
Other photos from `www.pixabay.com`.