Engineering Java 7's Dual Pivot Quicksort Using MaLiJAn

Sebastian Wild Markus E. Nebel Raphael Reitzig Ulrich Laube

[wild, nebel, r_reitzi, laube] @cs.uni-kl.de

Computer Science Department
University of Kaiserslautern

January 7, 2013
Meeting on Algorithm Engineering & Experiments 2013

Background

- Since Java 7: new **dual pivot Quicksort** in JRE library
- Basic algorithm by **Vladimir Yaroslavskiy**
- Optimizations by Jon Bentley, Joshua Bloch and others (see java.core-libs.devel mailing list)
  - Motivated by experience with classic Quicksort
  - Validated by **running time benchmark**

In this talk:

- Can we exploit special properties of **dual** pivot Quicksort?
- Can we get more insight than running time measurements?

. . . stay tuned

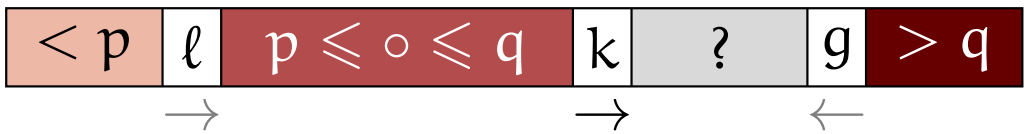
Java 7's Dual Pivot Quicksort – Example

Yaroslavskiy's Dual Pivot Quicksort
(used in Oracle's Java 7 `Arrays.sort(int[])`)



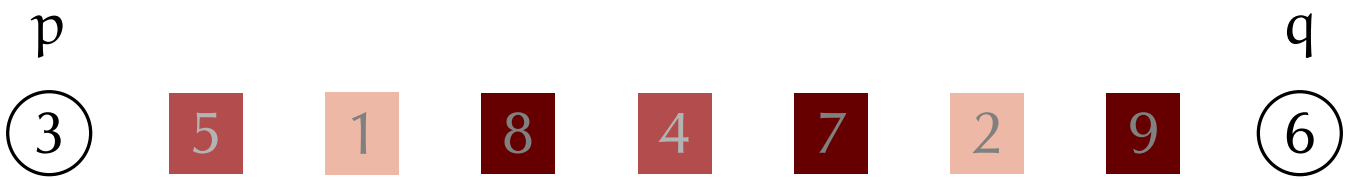
Select two elements as pivots.

Invariant:



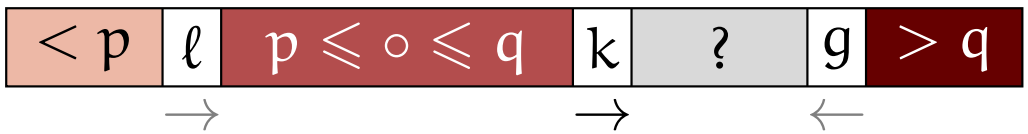
Java 7's Dual Pivot Quicksort – Example

Yaroslavskiy's Dual Pivot Quicksort
(used in Oracle's Java 7 `Arrays.sort(int[])`)



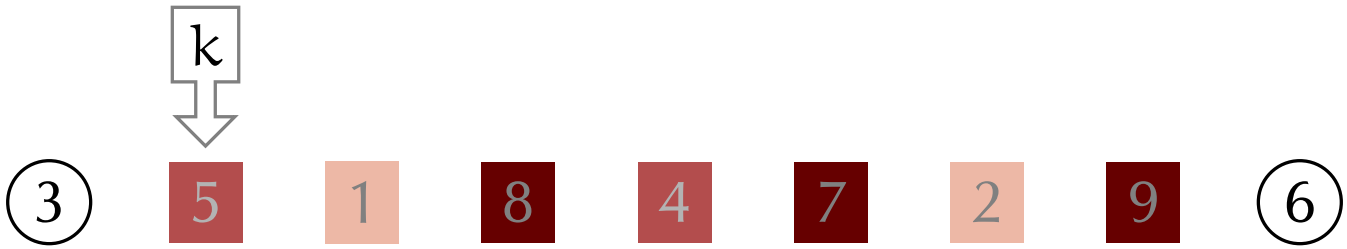
Only value relative to pivot counts.

Invariant:



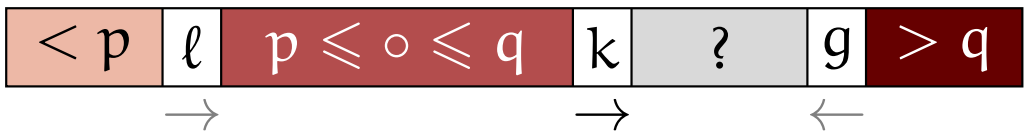
Java 7's Dual Pivot Quicksort – Example

Yaroslavskiy's Dual Pivot Quicksort
(used in Oracle's Java 7 `Arrays.sort(int[])`)



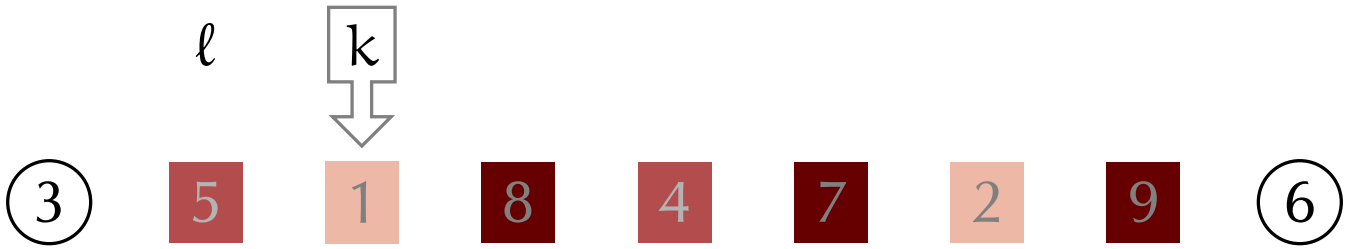
$A[k]$ is medium $\rightsquigarrow$ go on

Invariant:



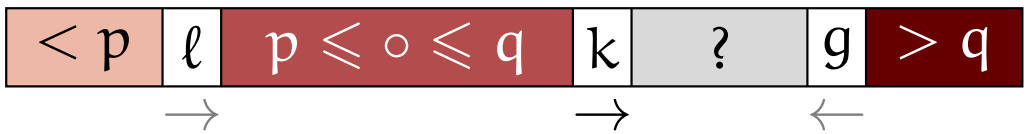
Java 7's Dual Pivot Quicksort – Example

Yaroslavskiy's Dual Pivot Quicksort
(used in Oracle's Java 7 `Arrays.sort(int[])`)



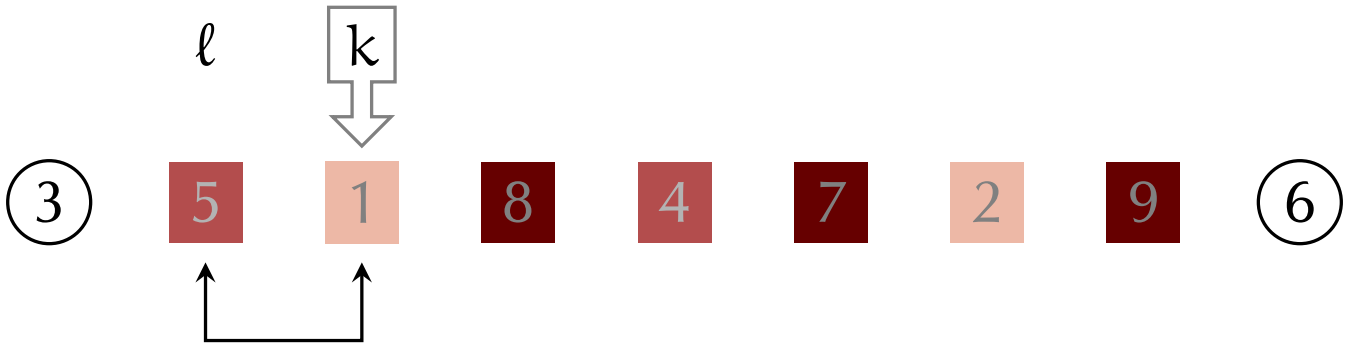
A[k] is small $\rightsquigarrow$ Swap to left

Invariant:



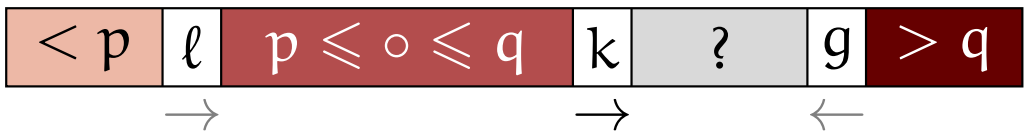
Java 7's Dual Pivot Quicksort – Example

Yaroslavskiy's Dual Pivot Quicksort
(used in Oracle's Java 7 `Arrays.sort(int[])`)



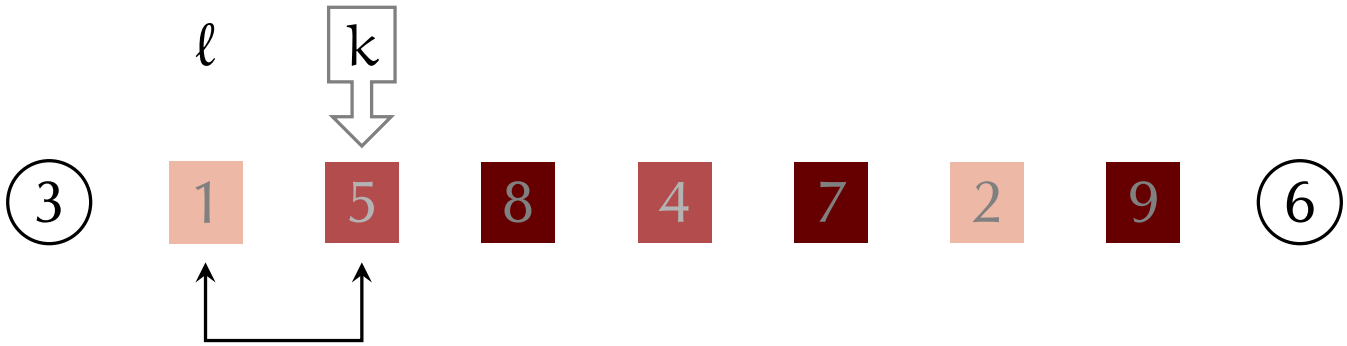
Swap small element to left end.

Invariant:



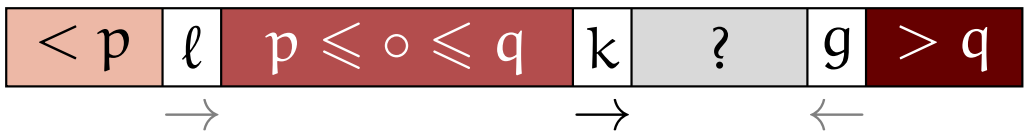
Java 7's Dual Pivot Quicksort – Example

Yaroslavskiy's Dual Pivot Quicksort
(used in Oracle's Java 7 `Arrays.sort(int[])`)



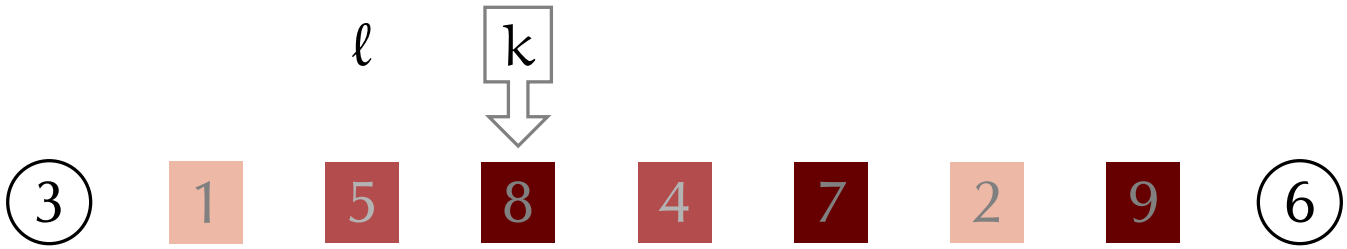
Swap small element to left end.

Invariant:



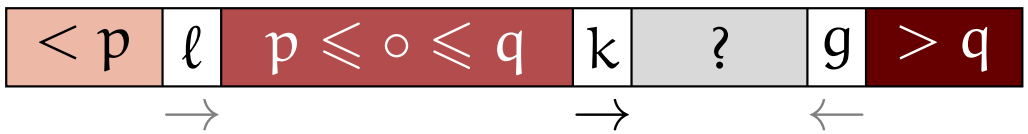
Java 7's Dual Pivot Quicksort – Example

Yaroslavskiy's Dual Pivot Quicksort
(used in Oracle's Java 7 `Arrays.sort(int[])`)



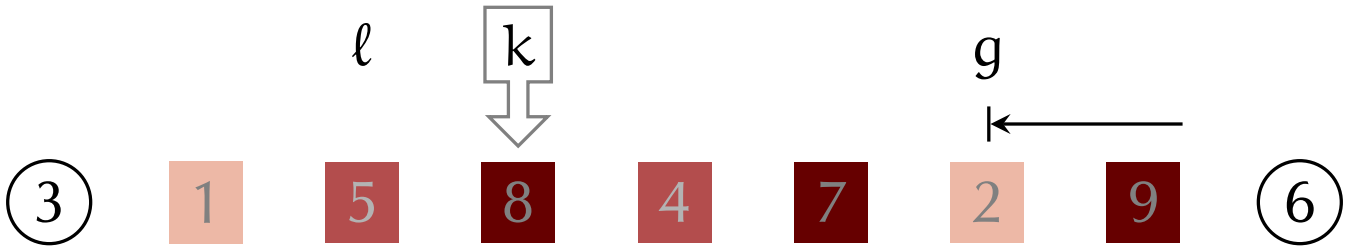
A[k] is large ⇝ Find swap partner.

Invariant:



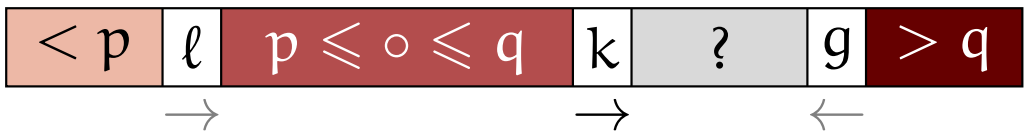
Java 7's Dual Pivot Quicksort – Example

Yaroslavskiy's Dual Pivot Quicksort
(used in Oracle's Java 7 `Arrays.sort(int[])`)



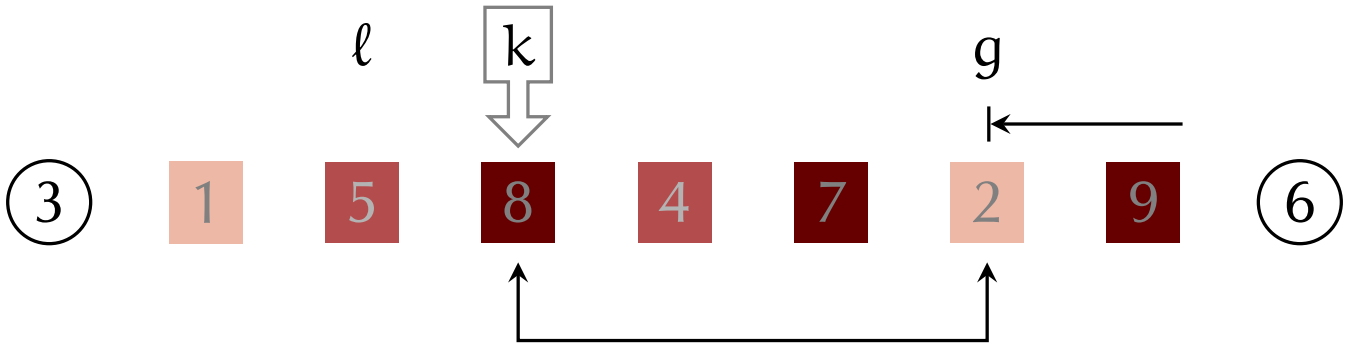
$A[k]$ is large $\rightsquigarrow$ Find swap partner:
g skips over large elements.

Invariant:



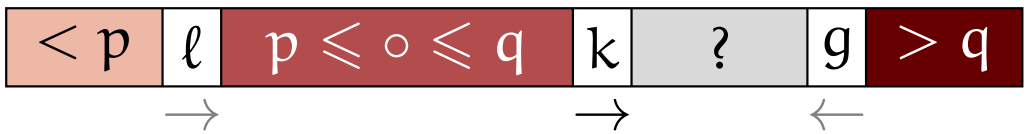
Java 7's Dual Pivot Quicksort – Example

Yaroslavskiy's Dual Pivot Quicksort
(used in Oracle's Java 7 `Arrays.sort(int[])`)



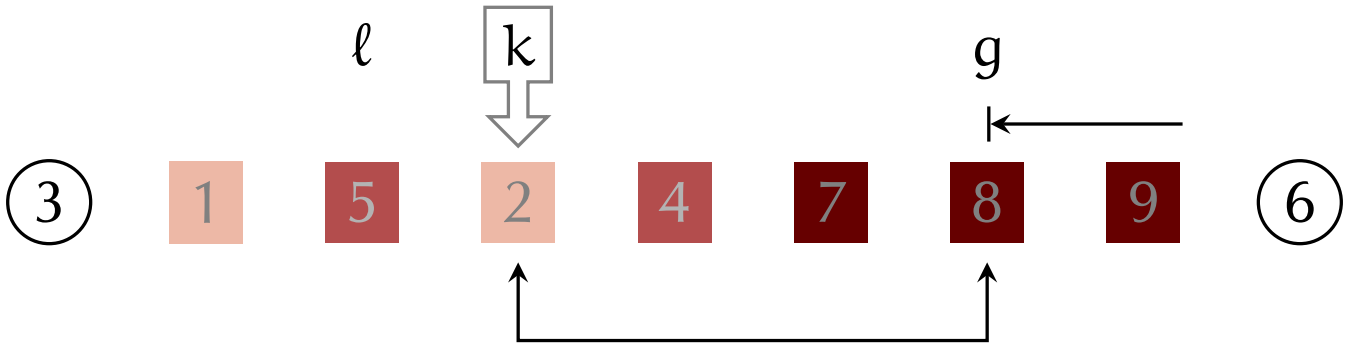
A[k] is large ⇝ Swap

Invariant:



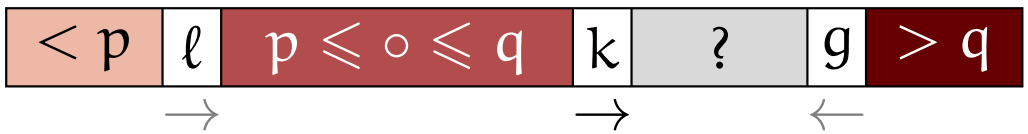
Java 7's Dual Pivot Quicksort – Example

Yaroslavskiy's Dual Pivot Quicksort
(used in Oracle's Java 7 `Arrays.sort(int[])`)



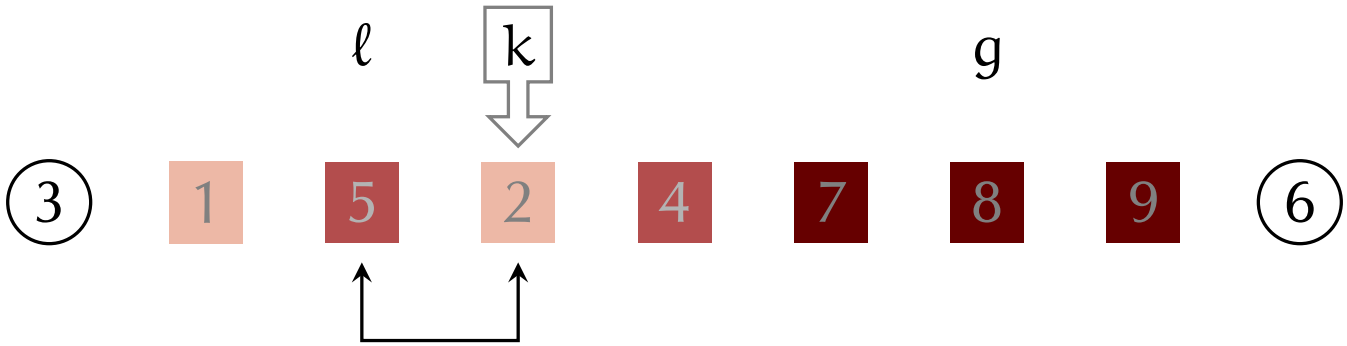
A[k] is large ⇝ Swap

Invariant:



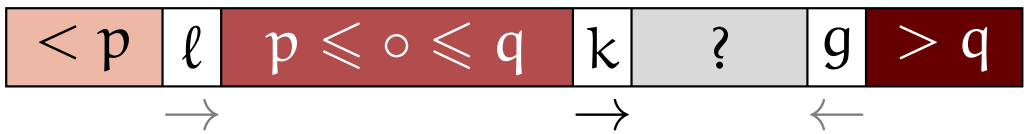
Java 7's Dual Pivot Quicksort – Example

Yaroslavskiy's Dual Pivot Quicksort
(used in Oracle's Java 7 `Arrays.sort(int[])`)



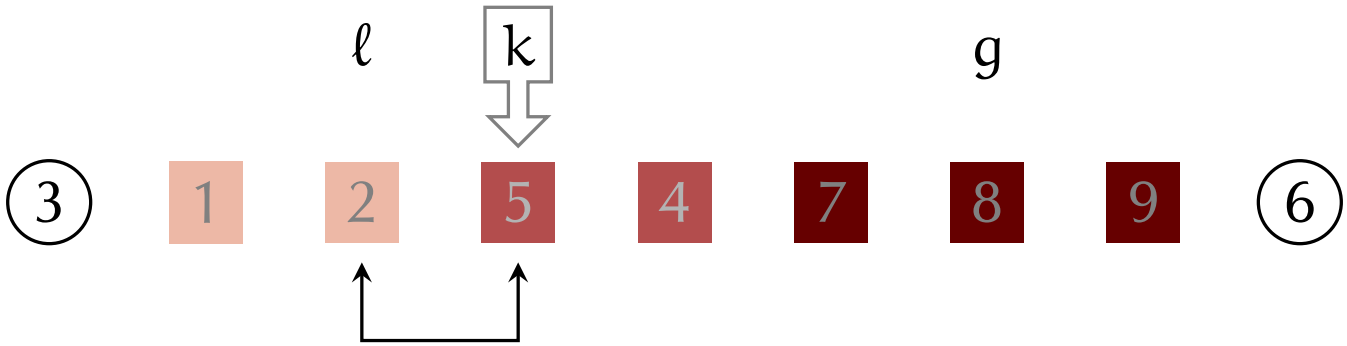
A[k] is old A[g], small $\rightsquigarrow$ Swap to left

Invariant:



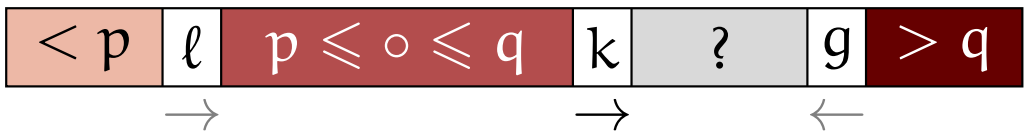
Java 7's Dual Pivot Quicksort – Example

Yaroslavskiy's Dual Pivot Quicksort
(used in Oracle's Java 7 `Arrays.sort(int[])`)



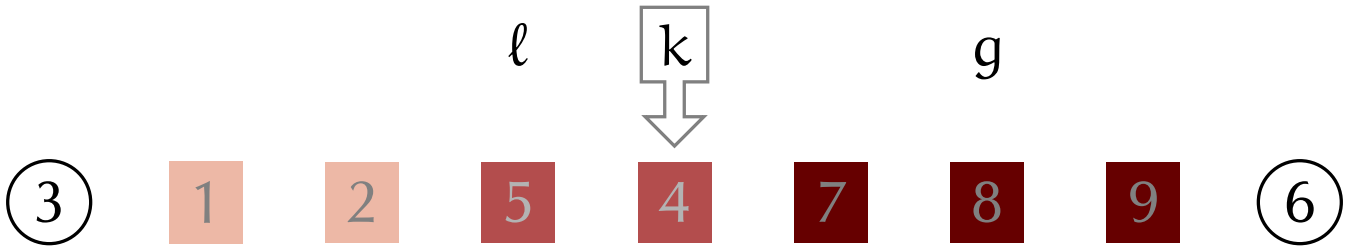
A[k] is old A[g], small $\rightsquigarrow$ Swap to left

Invariant:



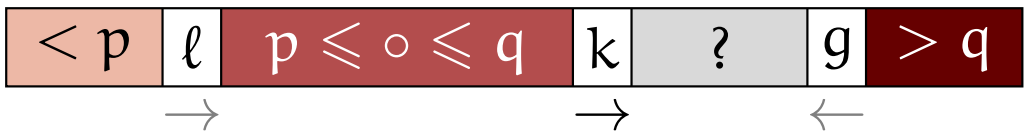
Java 7's Dual Pivot Quicksort – Example

Yaroslavskiy's Dual Pivot Quicksort
(used in Oracle's Java 7 `Arrays.sort(int[])`)



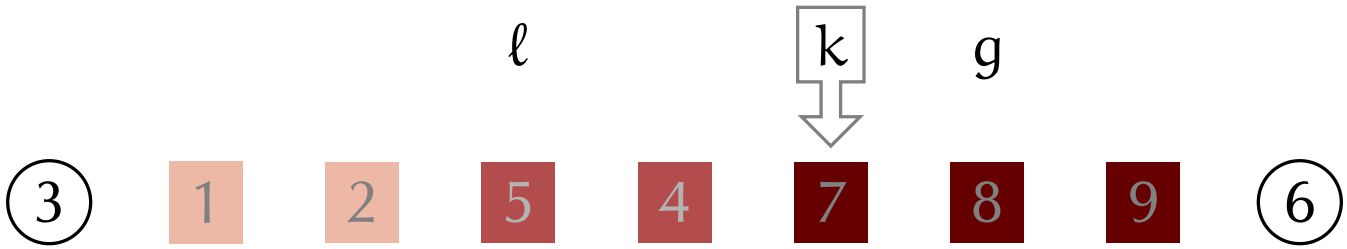
$A[k]$ is medium $\rightsquigarrow$ go on

Invariant:



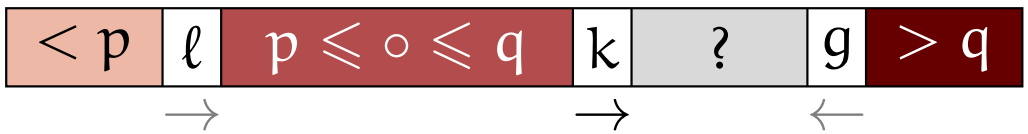
Java 7's Dual Pivot Quicksort – Example

Yaroslavskiy's Dual Pivot Quicksort
(used in Oracle's Java 7 `Arrays.sort(int[])`)



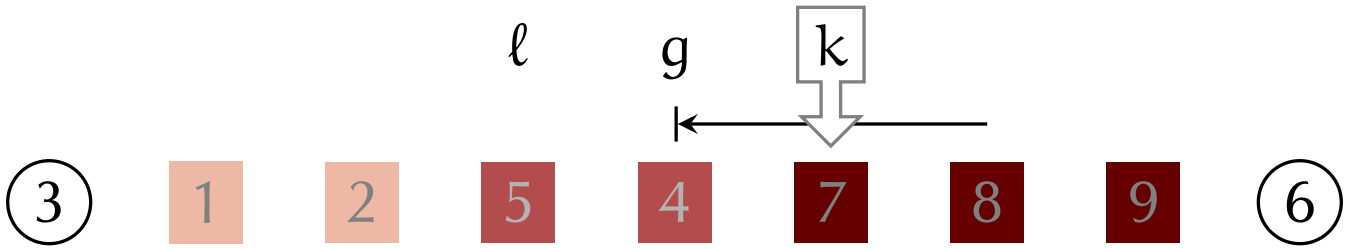
$A[k]$ is large $\rightsquigarrow$ Find swap partner.

Invariant:



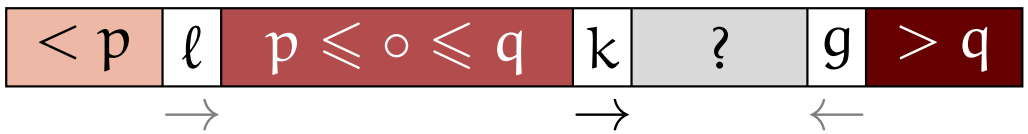
Java 7's Dual Pivot Quicksort – Example

Yaroslavskiy's Dual Pivot Quicksort
(used in Oracle's Java 7 `Arrays.sort(int[])`)



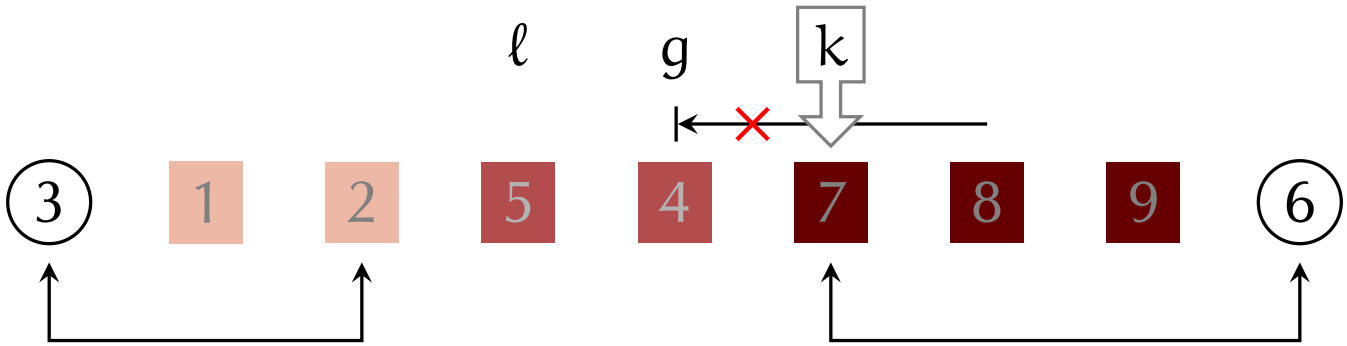
A[k] is large ⇝ Find swap partner:
g skips over large elements.

Invariant:



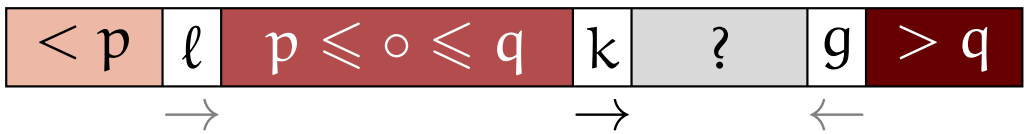
Java 7's Dual Pivot Quicksort – Example

Yaroslavskiy's Dual Pivot Quicksort
(used in Oracle's Java 7 `Arrays.sort(int[])`)



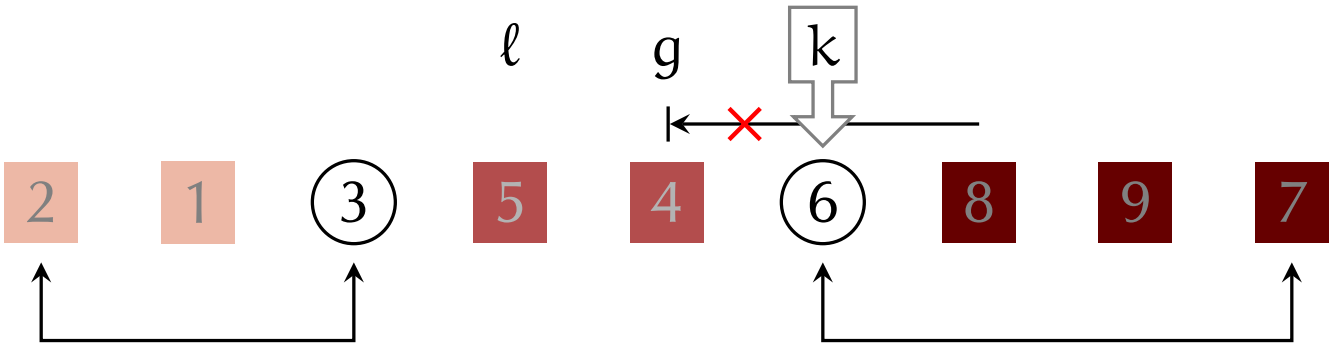
g and k have crossed!
Swap pivots in place

Invariant:



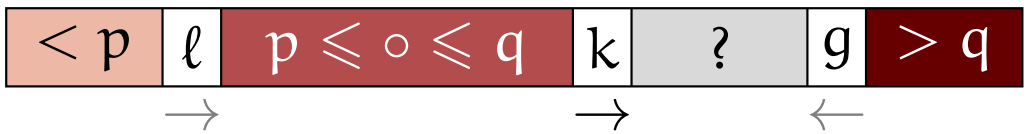
Java 7's Dual Pivot Quicksort – Example

Yaroslavskiy's Dual Pivot Quicksort
(used in Oracle's Java 7 `Arrays.sort(int[])`)



g and k have crossed!
Swap pivots in place

Invariant:



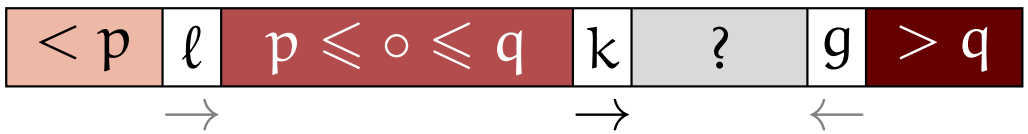
Java 7's Dual Pivot Quicksort – Example

Yaroslavskiy's Dual Pivot Quicksort
(used in Oracle's Java 7 `Arrays.sort(int[])`)



Partitioning done!

Invariant:



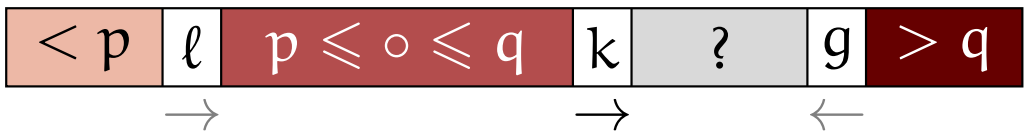
Java 7's Dual Pivot Quicksort – Example

Yaroslavskiy's Dual Pivot Quicksort
(used in Oracle's Java 7 `Arrays.sort(int[])`)



Recursively sort three sublists.

Invariant:



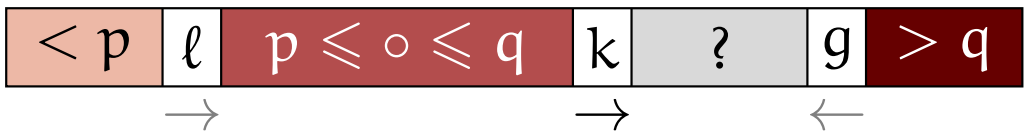
Java 7's Dual Pivot Quicksort – Example

Yaroslavskiy's Dual Pivot Quicksort
(used in Oracle's Java 7 `Arrays.sort(int[])`)

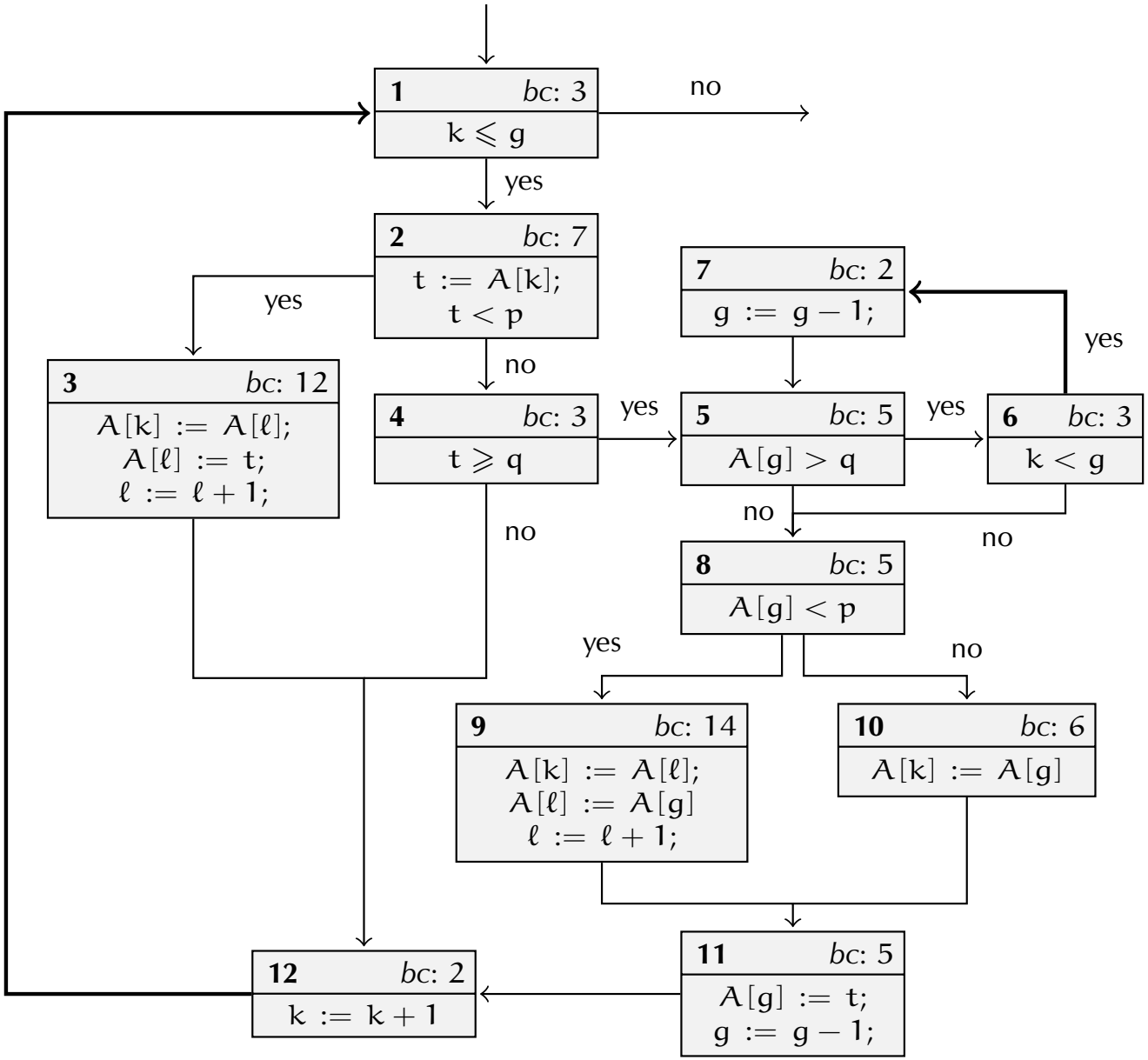


Done.

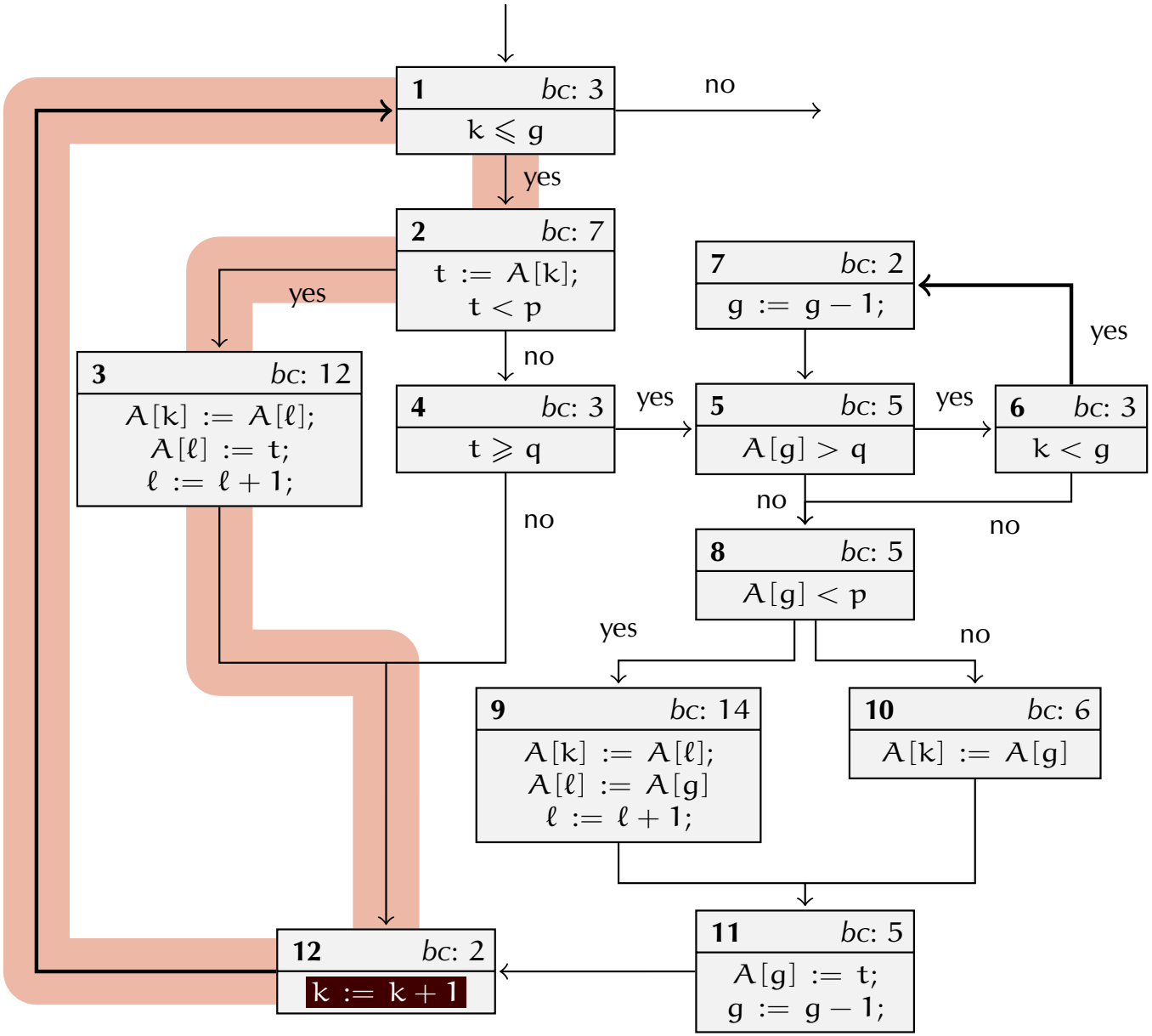
Invariant:



Control Flow Graph of Partitioning Loop



Control Flow Graph of Partitioning Loop



Cycle 1

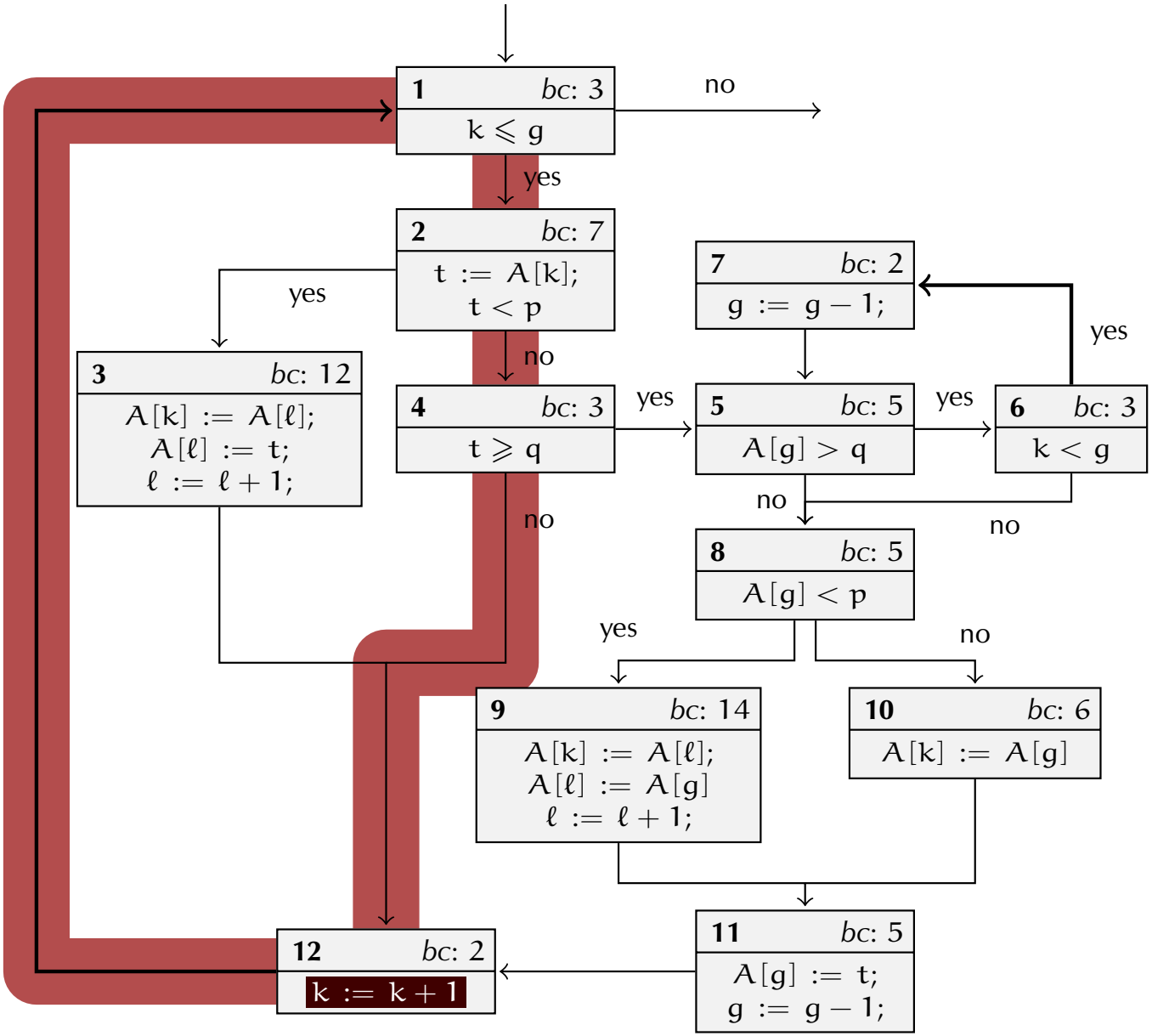
$A[k]$: small

$A[g]$: —

$\Delta(g - k)$: 1

Bytecode Instructions: 24

Control Flow Graph of Partitioning Loop



Cycle 2

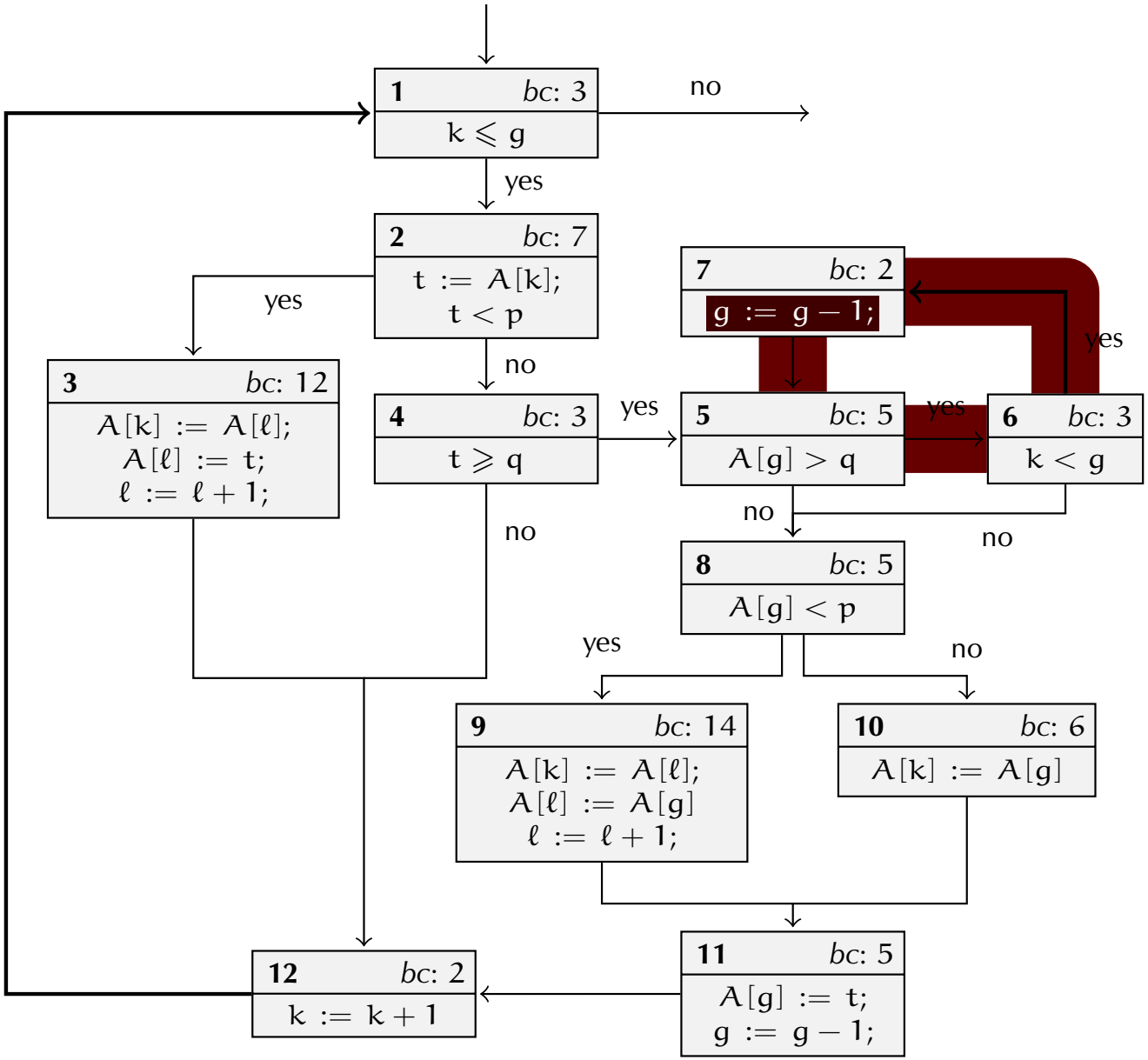
$A[k]$: medium

$A[g]$: —

$\Delta(g - k)$: 1

Bytecode Instructions: 15

Control Flow Graph of Partitioning Loop



Cycle 3

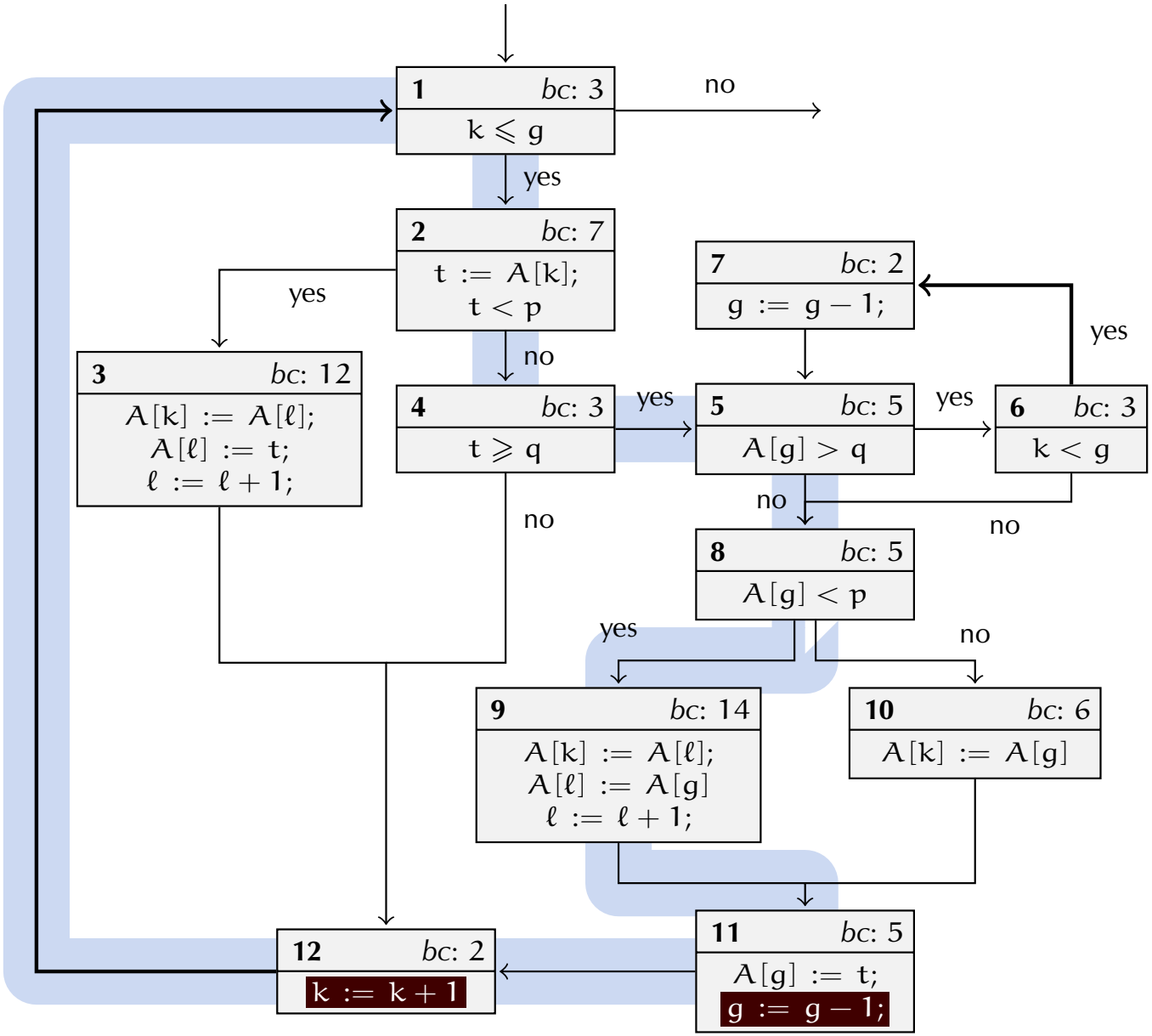
A[k]: large

A[g]: large

$\Delta(g - k)$: 1

Bytecode Instructions: 10

Control Flow Graph of Partitioning Loop



Cycle 4

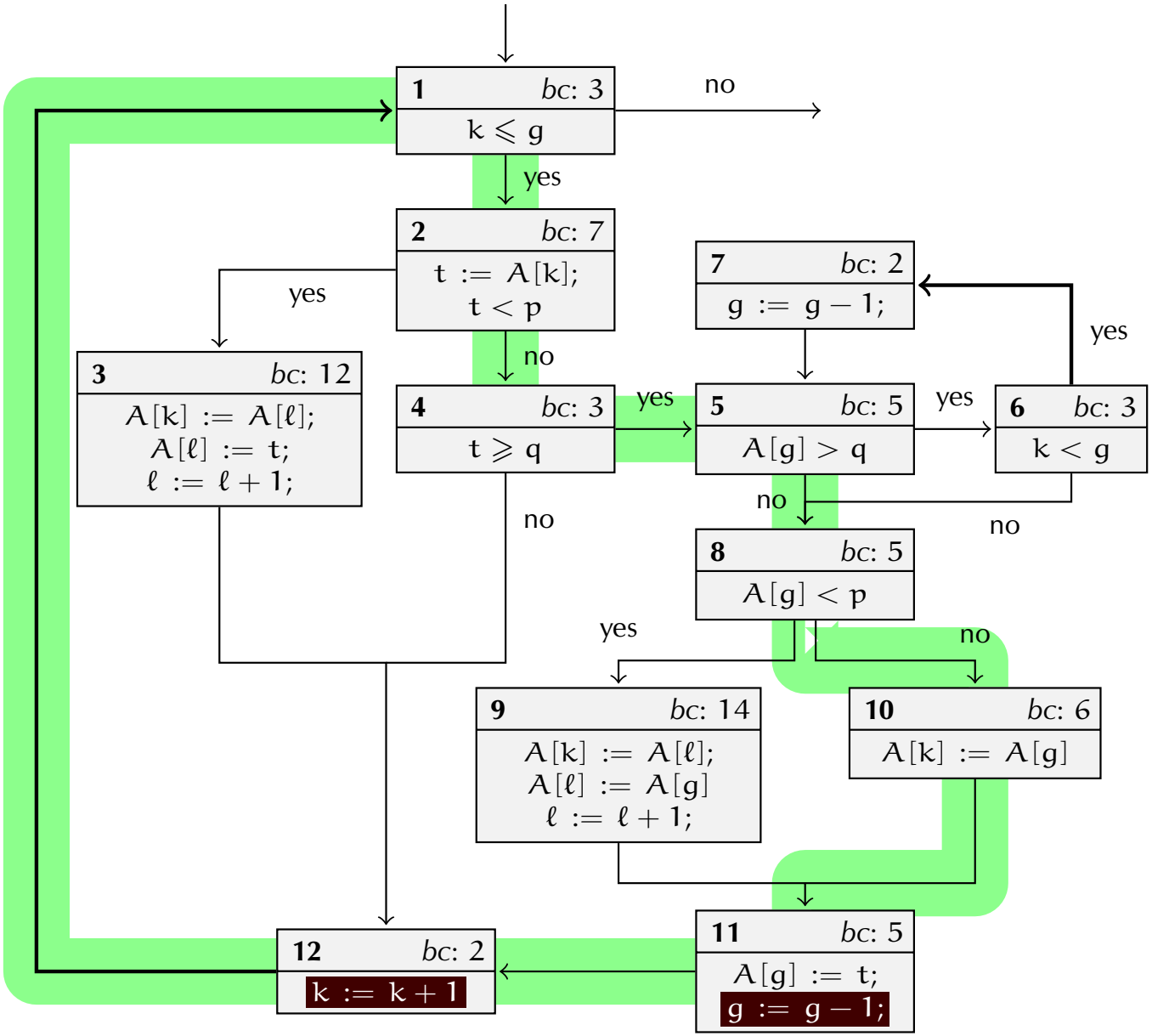
$A[k]$: large

$A[g]$: small

$\Delta(g - k)$: 2

Bytecode Instructions: 44

Control Flow Graph of Partitioning Loop



Cycle 5

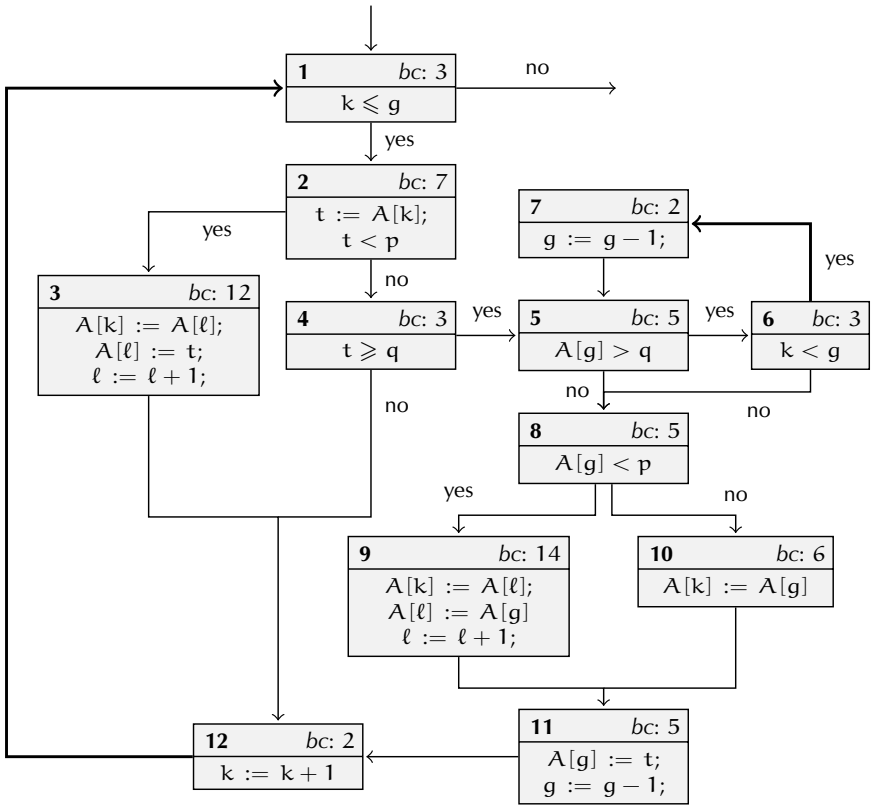
$A[k]$: large

$A[g]$: medium

$\Delta(g - k)$: 2

Bytecode Instructions: 36

Asymmetry



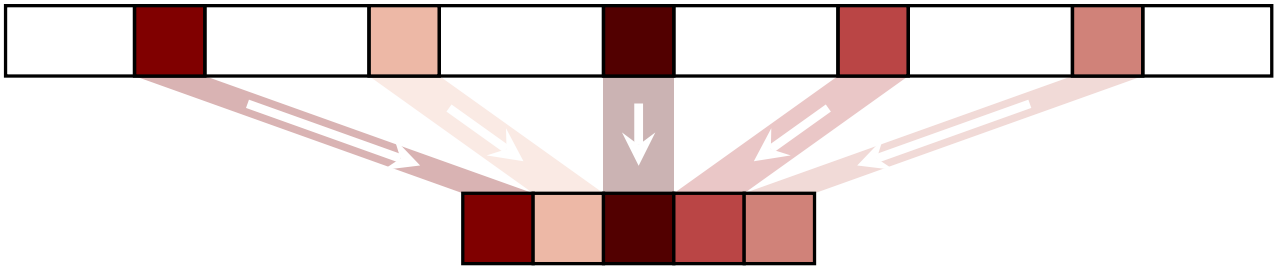
- Algorithm is **asymmetric:**
  - Cycles have different cost

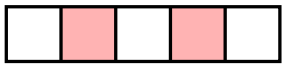
  ⇝ Would rather execute cheap ones often
- Cycles chosen by **classes** small, medium or large
- Probability for classes depends on **pivot values**

⇝ *Maybe we can "influence pivot values accordingly"?*

Pivot Sampling

- Well-known optimization for **classic** Quicksort: **median-of-three**
  $\rightsquigarrow$ pivot closer to **median** of whole list
- In **JRE7 Quicksort implementation**: natural extension for 2 pivots:



tertiles-of-five 

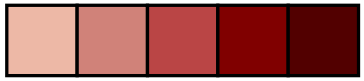
$\rightsquigarrow$ pivots closer to **tertiles** of whole list

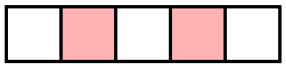
- **9** other possibilities to pick p and q out of 5 elements:

Pivot Sampling

- Well-known optimization for **classic** Quicksort: **median-of-three**
  $\rightsquigarrow$ pivot closer to **median** of whole list
- In **JRE7 Quicksort implementation**: natural extension for 2 pivots:





tertiles-of-five 

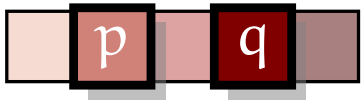
$\rightsquigarrow$ pivots closer to **tertiles** of whole list

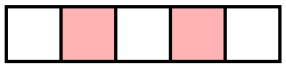
- **9** other possibilities to pick p and q out of 5 elements:

Pivot Sampling

- Well-known optimization for **classic** Quicksort: **median-of-three**
  $\rightsquigarrow$ pivot closer to **median** of whole list
- In **JRE7 Quicksort implementation**: natural extension for 2 pivots:





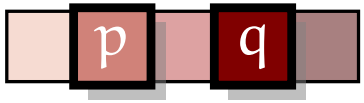
tertiles-of-five 
$\rightsquigarrow$ pivots closer to **tertiles** of whole list

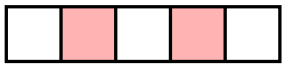
- **9** other possibilities to pick p and q out of 5 elements:

Pivot Sampling

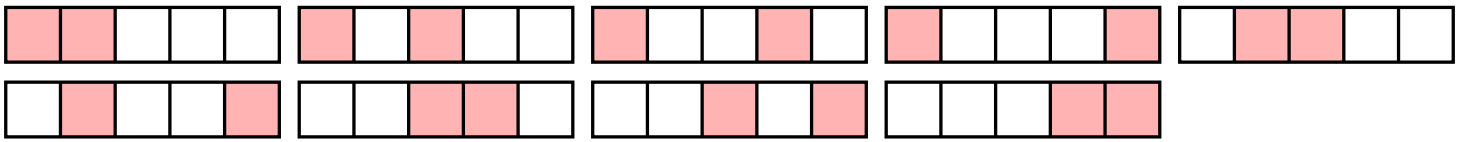
- Well-known optimization for **classic** Quicksort: **median-of-three**
  $\rightsquigarrow$ pivot closer to **median** of whole list
- In **JRE7 Quicksort implementation**: natural extension for 2 pivots:



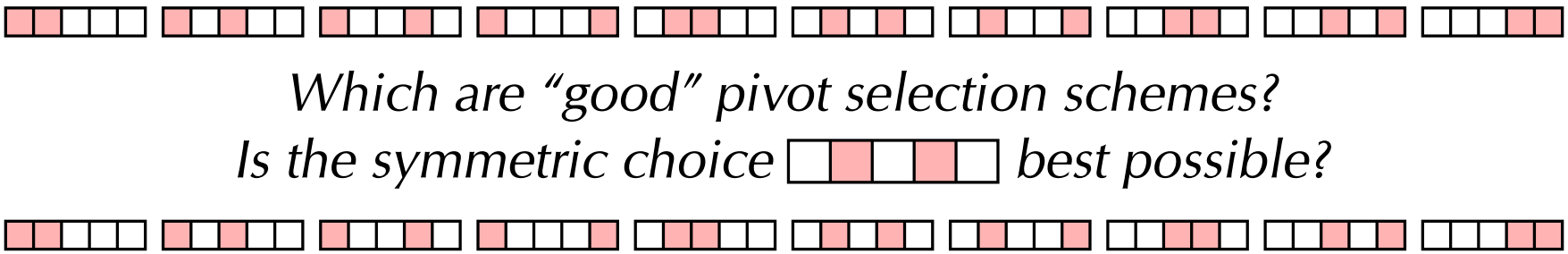


tertiles-of-five 
$\rightsquigarrow$ pivots closer to **tertiles** of whole list

- **9** other possibilities to pick p and q out of 5 elements:



Optimizing Pivot Sampling

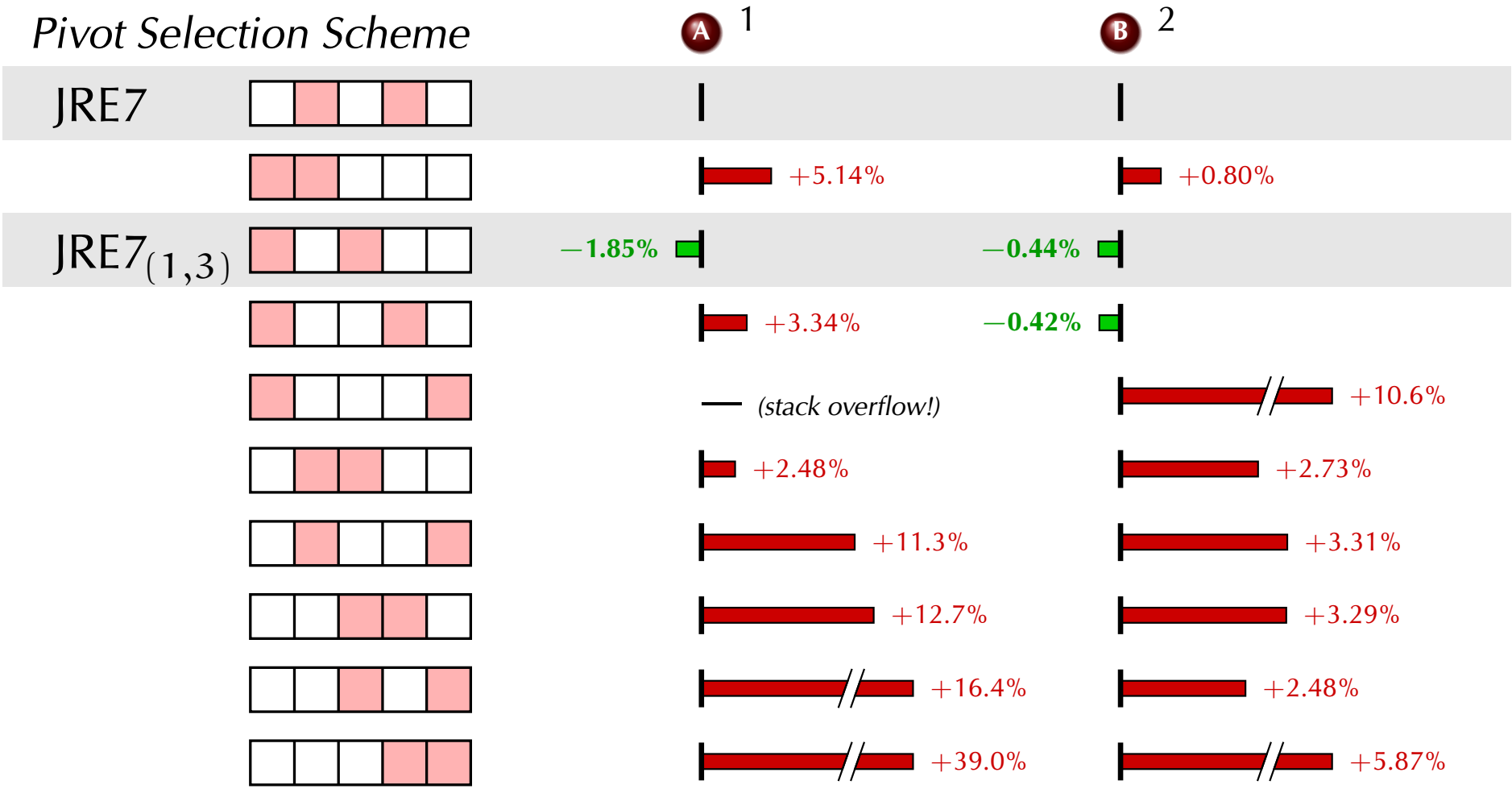


Which are "good" pivot selection schemes?
Is the symmetric choice *best possible?*

- Need objective function to optimize
- Typical approaches to judge efficiency:
  - **A** Count number of **basic operations**.
    (Here: number of executed **Java Bytecode instructions**.)
  - **B** Measure total **running time**.

Optimizing Pivot Sampling

Relative performance of pivot sampling compared to tertiles-of-five:

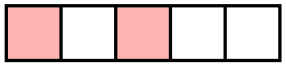
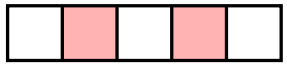


[1] Average number of executed bytecodes on almost sorted lists of length 10^5.

[2] Average running time on random permutations of length 10^6.

Methods

Model and Method

What *made JRE7*$_{(1,3)}$  *faster than JRE7* *?*

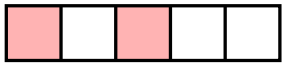
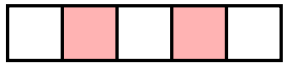
- . . . hard to tell from **total** time/bytecodes.
- Need a more **detailed model** of the program.

Idea: Decompose along control flow graph!

- View program as **Markov chain** over blocks
- Termination via absorbing state
- Transition $i \to j$ has **probability** $p_{i \to j}^{(n)}$ **depending on input size** n
- Visiting block i incurs **constant costs** $c(i)$
- Total cost is sum of block costs

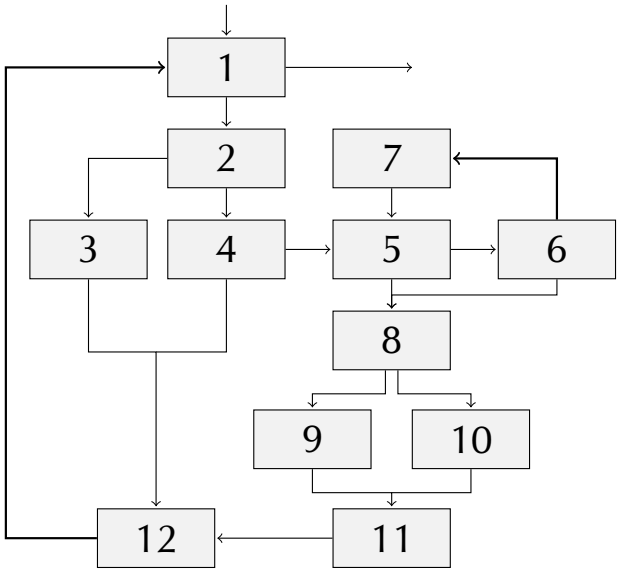
- Expected costs of program = expected costs of run of Markov chain
- Latter easy to compute

Model and Method

What *made JRE7*$_{(1,3)}$  *faster than JRE7* *?*

- . . . hard to tell from **total** time/bytecodes.
- Need a more **detailed model** of the program.

Idea: Decompose along control flow graph!



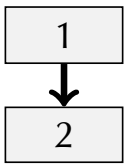
- View program as **Markov chain** over blocks
- Termination via absorbing state
- Transition $i \to j$ has **probability** $p_{i \to j}^{(n)}$ **depending on input size** n
- Visiting block i incurs **constant costs** $c(i)$
- Total cost is sum of block costs

- Expected costs of program = expected costs of run of Markov chain
- Latter easy to compute

Maximum Likelihood Analysis

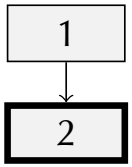
How to determine block costs and transition probabilities?

Transition Probabilities



- **Count** transitions in executions on **sample data**
  $\leadsto$ Allows arbitrary input distributions!
- Take relative frequency as estimate for $p_{i\to j}^{(n)}$
- **Extrapolate** $p_{i\to j}^{(n)}$ to a function $p_{i\to j}(n)$ in n

Block Costs



We consider two cost measures:

A. $bc(i)$ = number of **Bytecodes instructions** in block i.

B. $t(i)$ = **running time** of block i

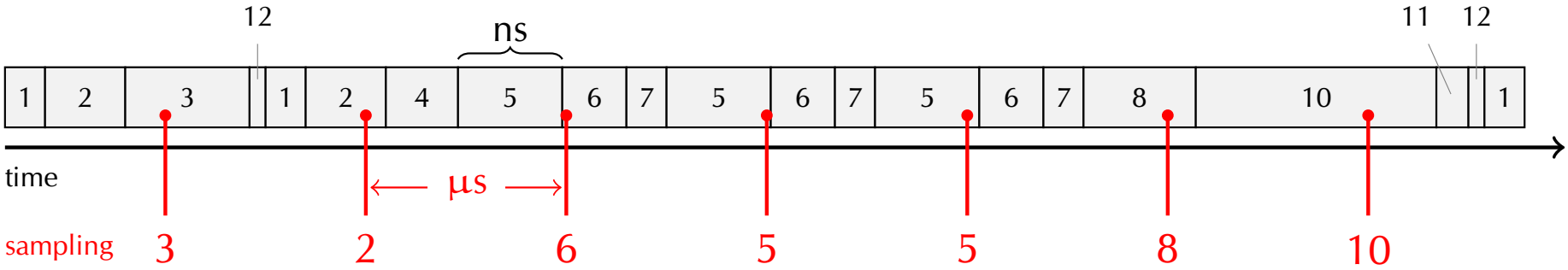
- All steps are automated in our tool MaLiJAn[3]

[3] `http://wwwagak.cs.uni-kl.de/malijan.html`

Block Sampling

- Running times $t(i)$ in B are typically few **nanoseconds**
  $\rightsquigarrow$ direct measurement not possible.

Idea: Sampling Based Approach

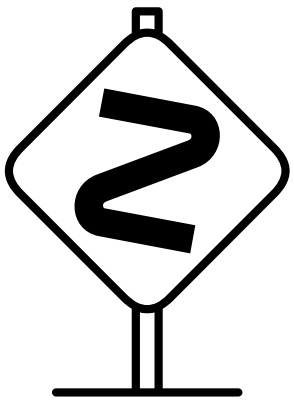


- In regular intervals, store **current** basic block (concurrently)
- We observe only $\approx 1‰$ of all blocks $\rightsquigarrow$ repeat execution

Relative **frequencies of observed samples** approach
relative **running time contribution** of blocks.

- Count in separate run how often block i gets executed in total
- Together, this allows to compute $t(i)$

A Decent Word of Caution



1. Determining current block adds a small **systematic** error.
2. **Java Specialty:** Just-in-time Compilation
   - Running time heavily influenced by HotSpot JIT compiler
   - JIT collects profiling information at beginning

   $\rightsquigarrow$ First input determines which optimizations are found

... more details in the paper

Input Distributions

We consider 2 different input distributions:

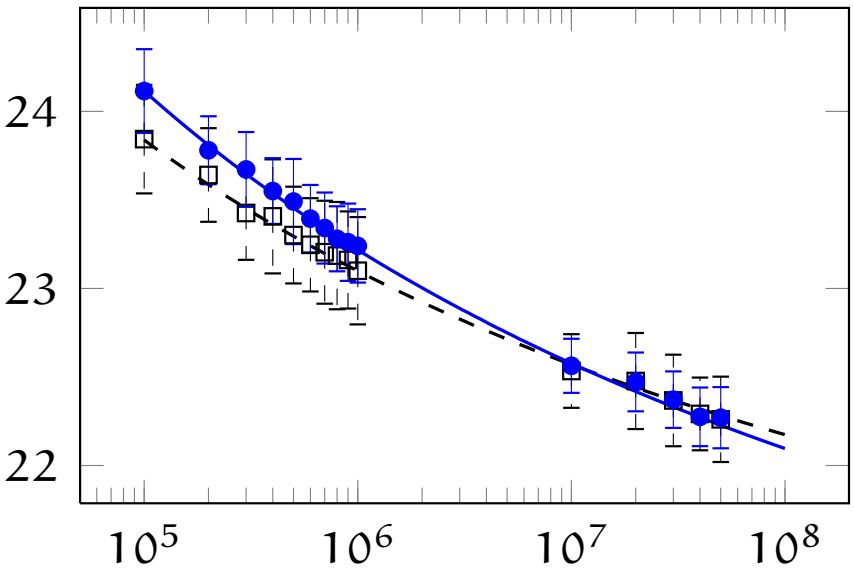
1. **Random Permutations**
   - well-studied in literature
2. **Almost Sorted Lists**
   - Random model by Brodal et al.[4]:
     $A[i]$ chosen i. i. d. uniform in $[i - d, i + d]$
     for constant d (here $d = 100$)

[4] G. Brodal, R. Fagerberg, G. Moruz: *On the Adaptiveness of Quicksort*, J. Exp. Algorithmics 12 (2008), pp. 3.2:1–3.2:20

Results

Asymptotic Expected Costs

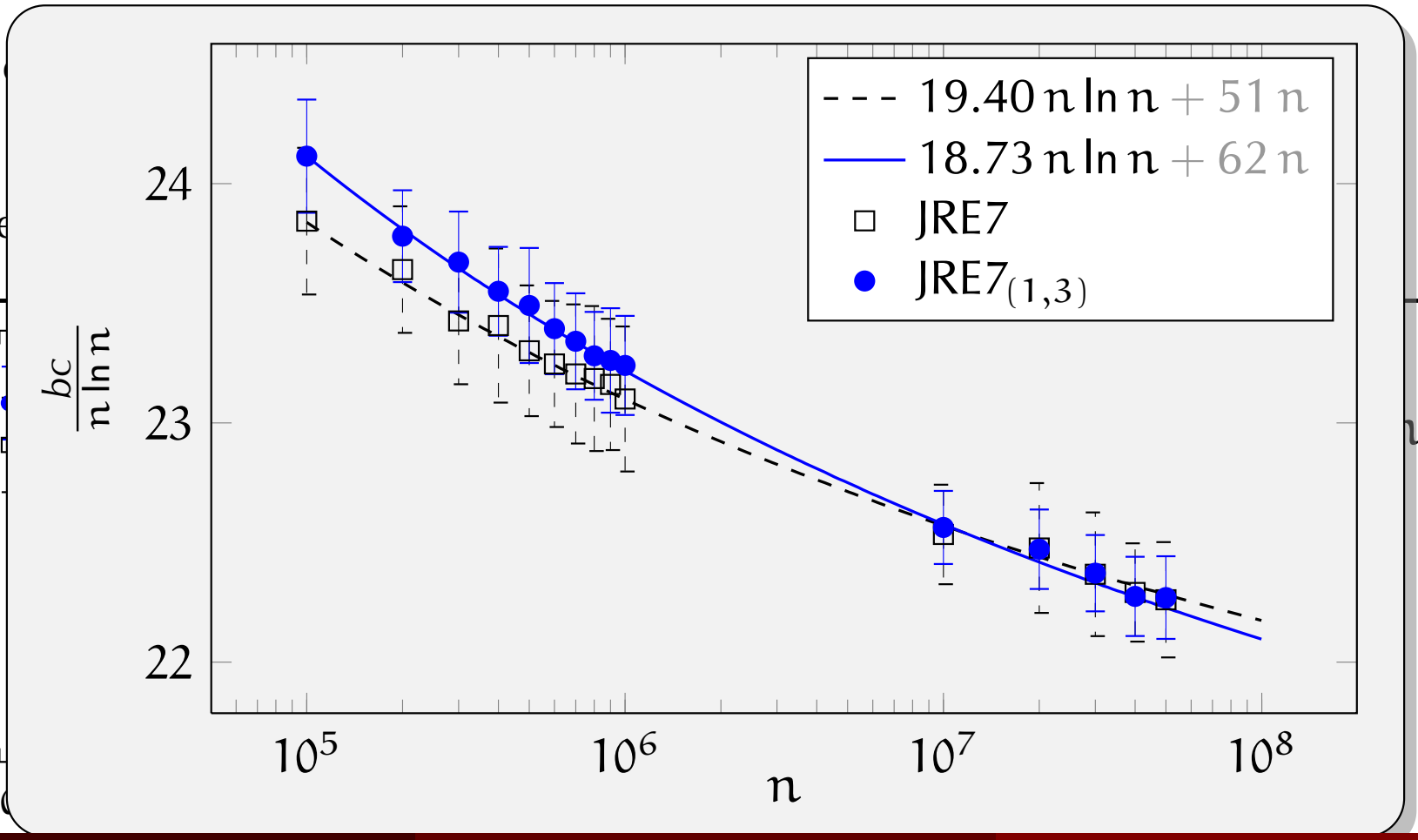
Measure		Algorithm	Random Permutations	Almost Sorted Lists
Bytecodes	A	□–□ JRE7 ●–● $\text{JRE7}_{(1,3)}$	$19.40\,n\ln n + 51\,n$ $18.73\,n\ln n + 62\,n$	
time -Xcomp	B	□–□ JRE7 ●–● $\text{JRE7}_{(1,3)}$		
time warmup	B	□–□ JRE7 ●–● $\text{JRE7}_{(1,3)}$		



- log. plot, normalized by $n\ln n$
- □- -□ JRE7, ●–● $\text{JRE7}_{(1,3)}$
- model fits data well!

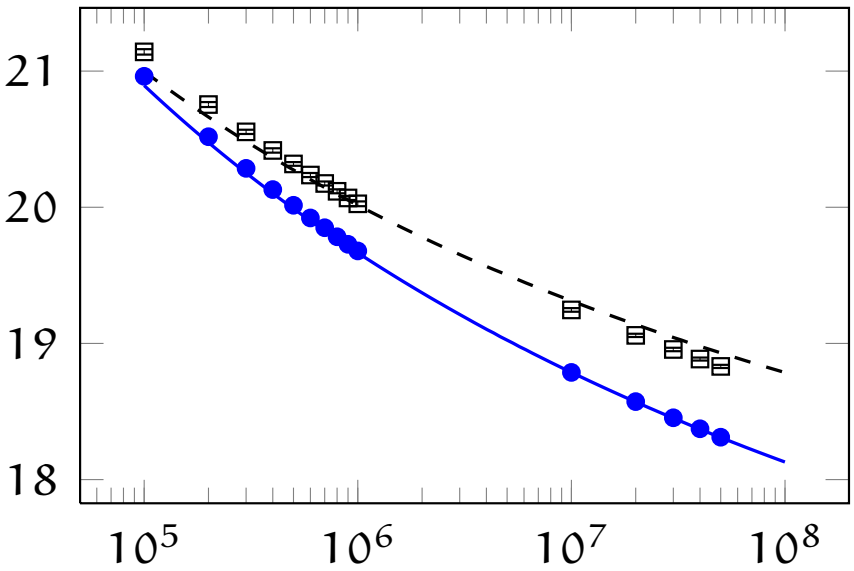
Asymptotic Expected Costs

Measure	Algorithm	Random Permutations	Almost Sorted Lists
Bytecodes (A)	JRE7	$19.40\,n\ln n + 51\,n$	
	$\text{JRE7}_{(1,3)}$	$18.73\,n\ln n + 62\,n$	



Asymptotic Expected Costs

Measure		Algorithm	Random Permutations	Almost Sorted Lists
Bytecodes	A	□–□ JRE7	$19.40\,n\ln n + 51\,n$	$15.10\,n\ln n + 68\,n$
		●–● $\text{JRE7}_{(1,3)}$	$18.73\,n\ln n + 62\,n$	$13.52\,n\ln n + 85\,n$
time `-Xcomp`	B	□–□ JRE7		
		●–● $\text{JRE7}_{(1,3)}$		
time warmup	B	□–□ JRE7		
		●–● $\text{JRE7}_{(1,3)}$		



- log. plot, normalized by $n\ln n$
- □- -□ JRE7, ●–● $\text{JRE7}_{(1,3)}$
- model fits data well!

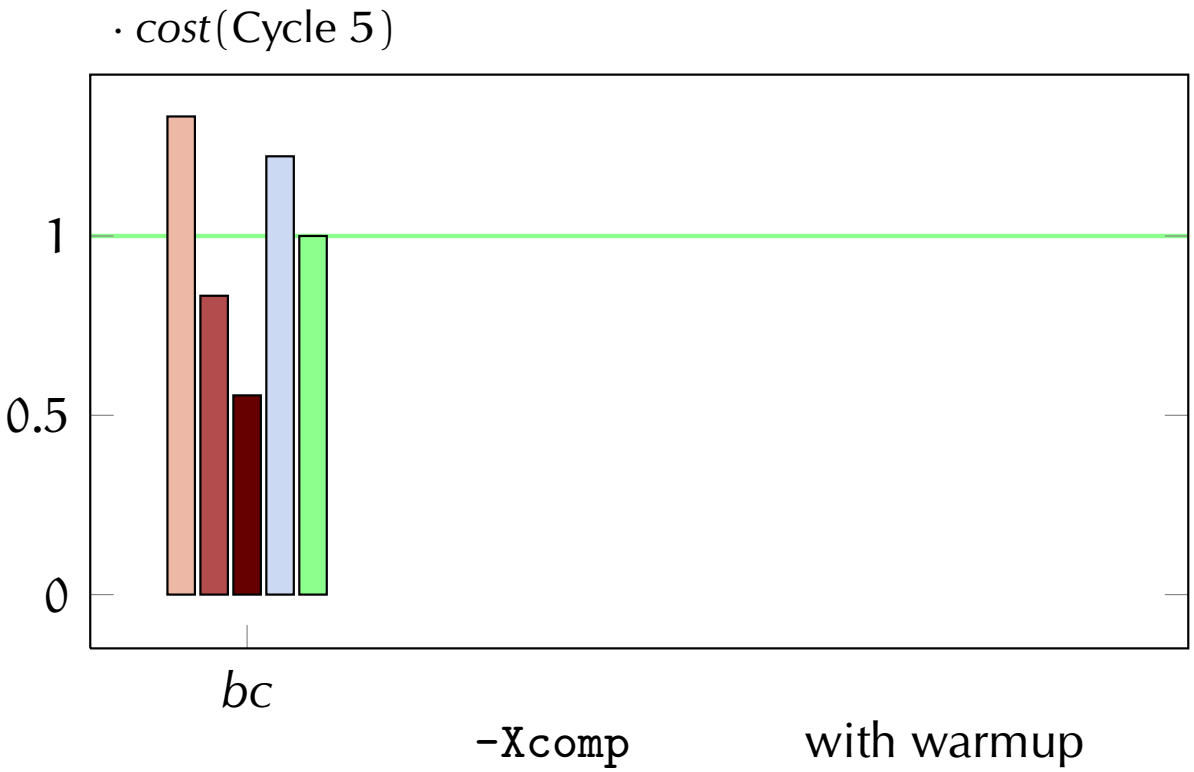
Asymptotic Expected Costs

Measure		Algorithm	Random Permutations	Almost Sorted Lists
Bytecodes	A	□–□ JRE7	$19.40\,n\ln n + 51\,n$	$15.10\,n\ln n + 68\,n$
		●–● $\text{JRE7}_{(1,3)}$	$18.73\,n\ln n + 62\,n$	$13.52\,n\ln n + 85\,n$
time `-Xcomp`	B	□–□ JRE7		
		●–● $\text{JRE7}_{(1,3)}$		
time warmup	B	□–□ JRE7		
		●–● $\text{JRE7}_{(1,3)}$		

$\rightsquigarrow$ asymptotically, $\text{JRE7}_{(1,3)}$ executes **less** **Bytecodes**!

*Can we explain, **why**?*

Cycle Costs



-Xcomp with warmup

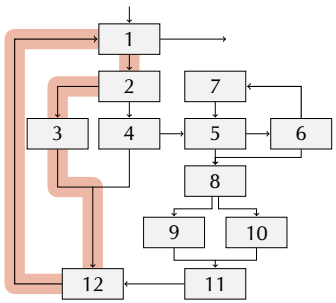
In #Bytecodes:

Cycle 3 **cheapest**

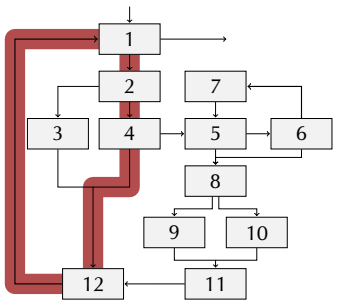
Cycle 1 **most expensive**

of all cycles

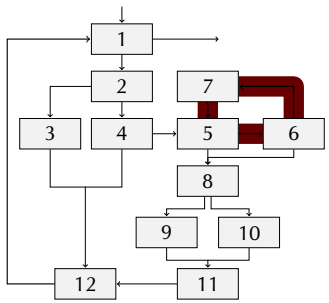
Cycle 1



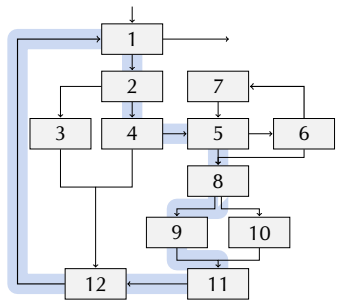
Cycle 2



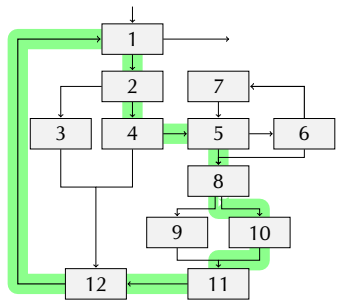
Cycle 3



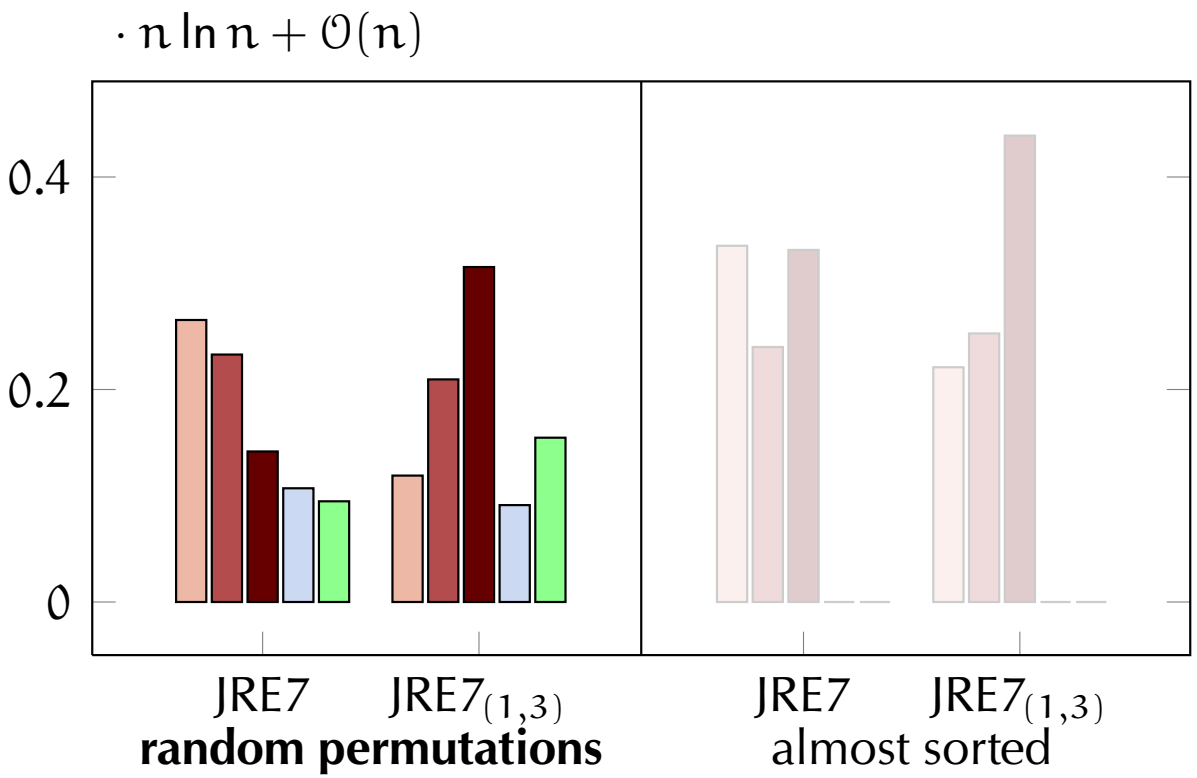
Cycle 4



Cycle 5



Asymptotic Cycle Frequencies



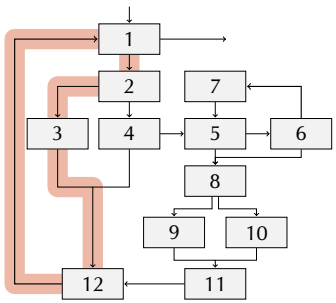
$\rightsquigarrow$ $\text{JRE7}_{(1,3)}$ executes

Cycle 3 **more** often

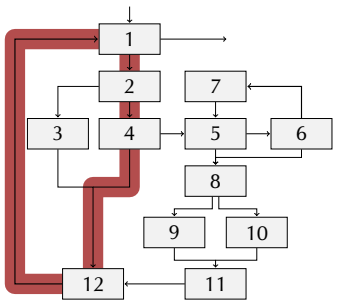
Cycle 1 **less** often

than JRE7

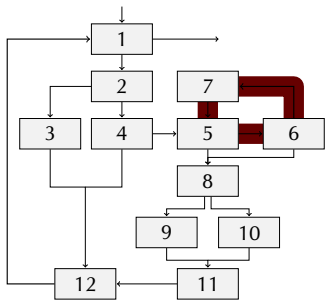
Cycle 1



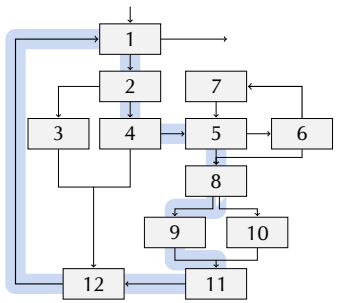
Cycle 2



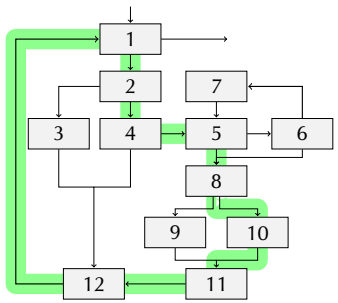
Cycle 3



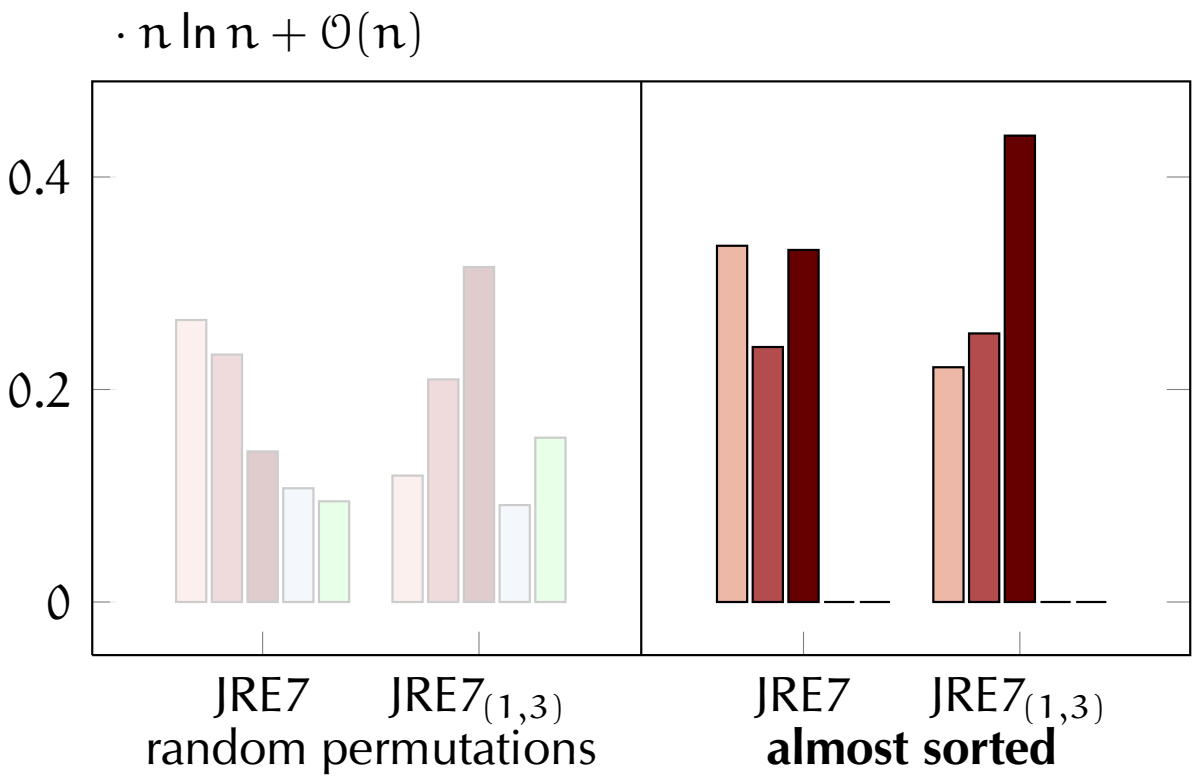
Cycle 4



Cycle 5



Asymptotic Cycle Frequencies



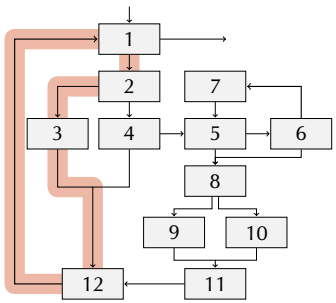
$\rightsquigarrow$ JRE7$_{(1,3)}$ executes

Cycle 3 **more** often

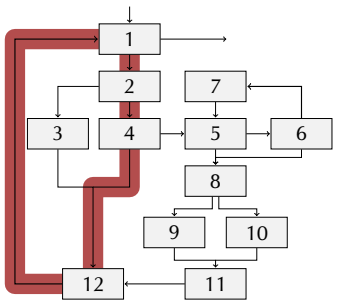
Cycle 1 **less** often

than JRE7

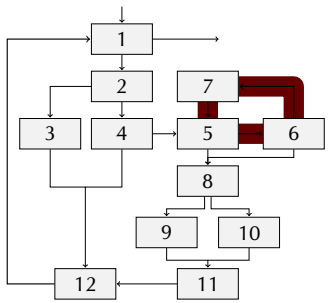
Cycle 1



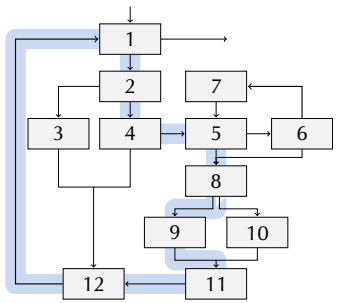
Cycle 2



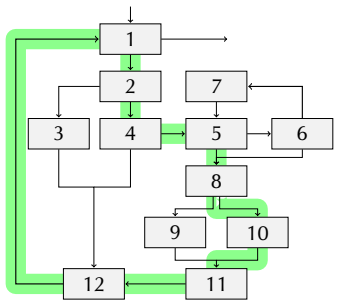
Cycle 3



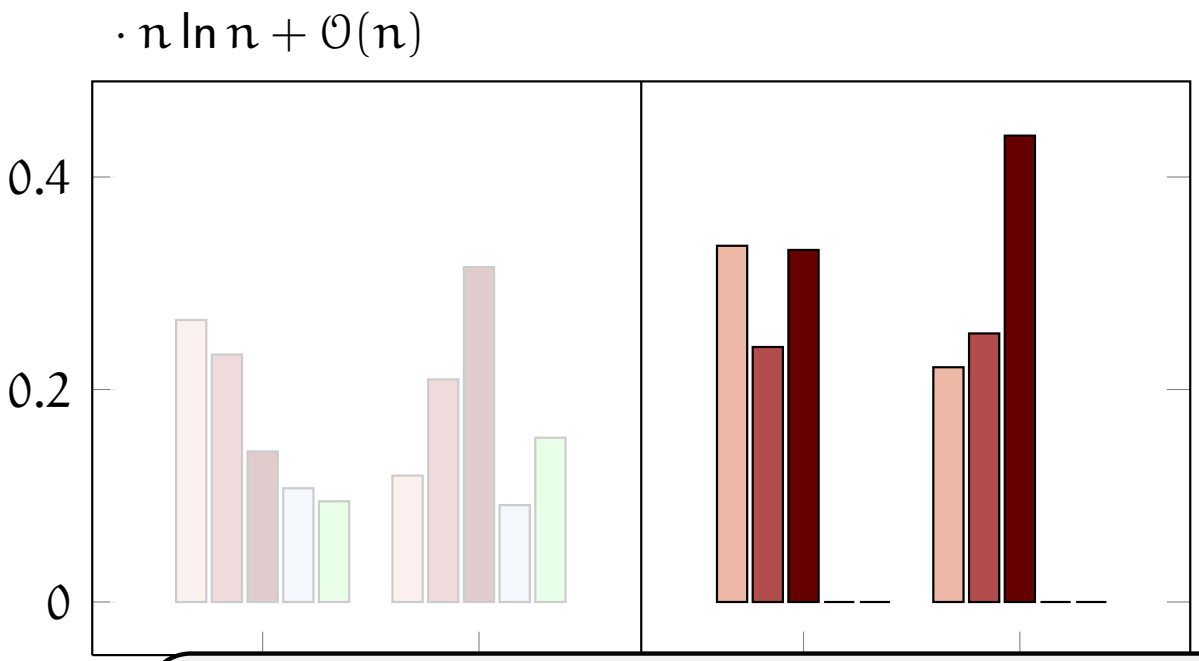
Cycle 4



Cycle 5



Asymptotic Cycle Frequencies



$\rightsquigarrow$ $\text{JRE7}_{(1,3)}$ executes

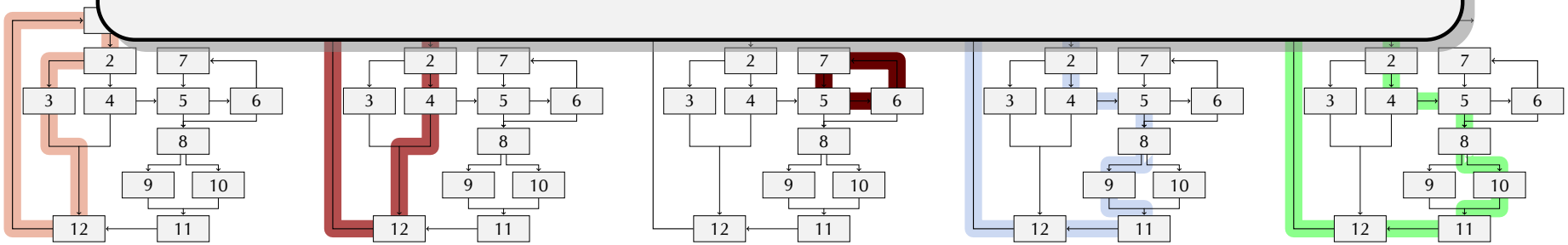
Cycle 3 **more** often

Cycle 1 **less** often

than JRE7

$\text{JRE7}_{(1,3)}$ executes cheap Cycle 3 more often and expensive Cycle 1 less often than JRE7.

$\rightsquigarrow$ **Asymptotically, less executed Bytecodes!**



Running Time Results

How about running time?

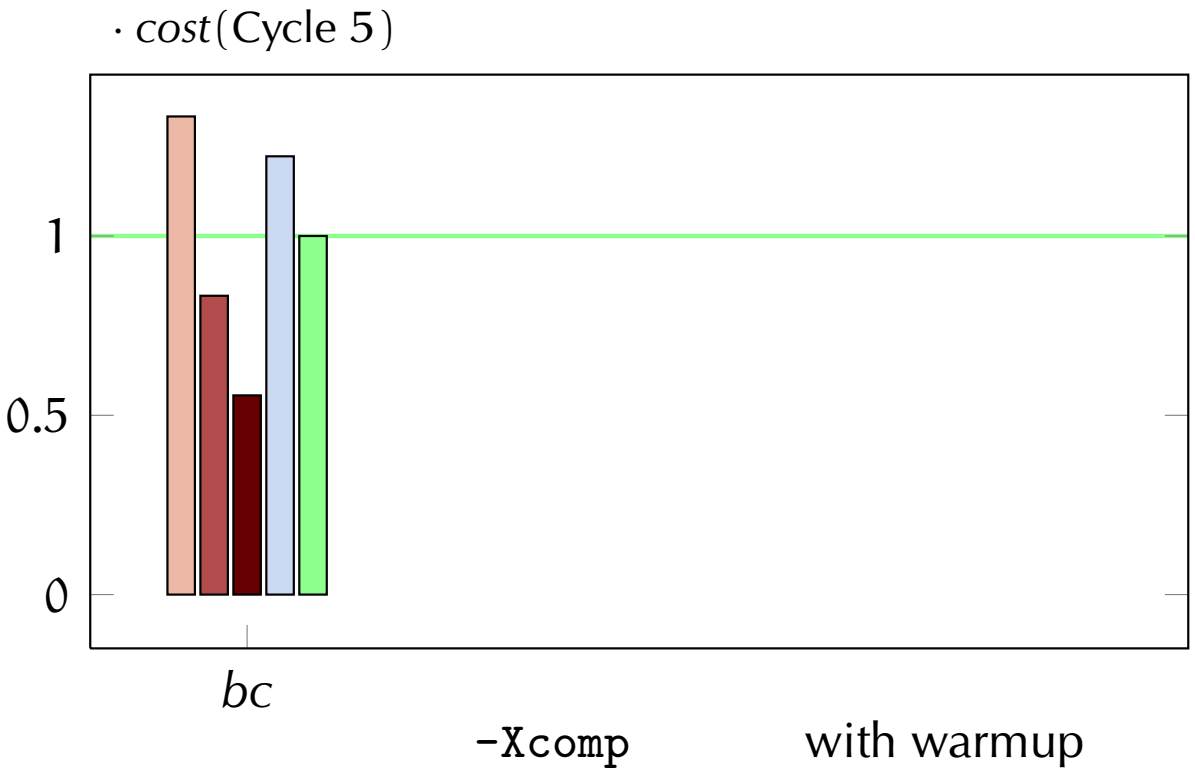
- HotSpot JIT compiler has two **modes**
  - `-Xcomp` JIT compiler **without** profiling information
  - warmup **profiling** JIT with **warmup** on fixed input
    $\rightsquigarrow$ trigger JIT compilation

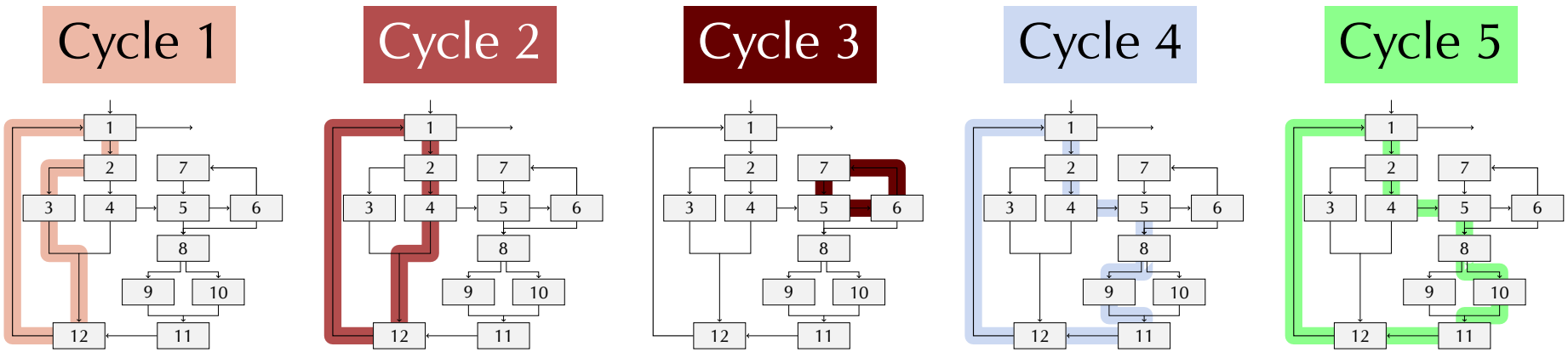
$\rightsquigarrow$ Do Block Sampling for both modes

Should we expect same block running times?

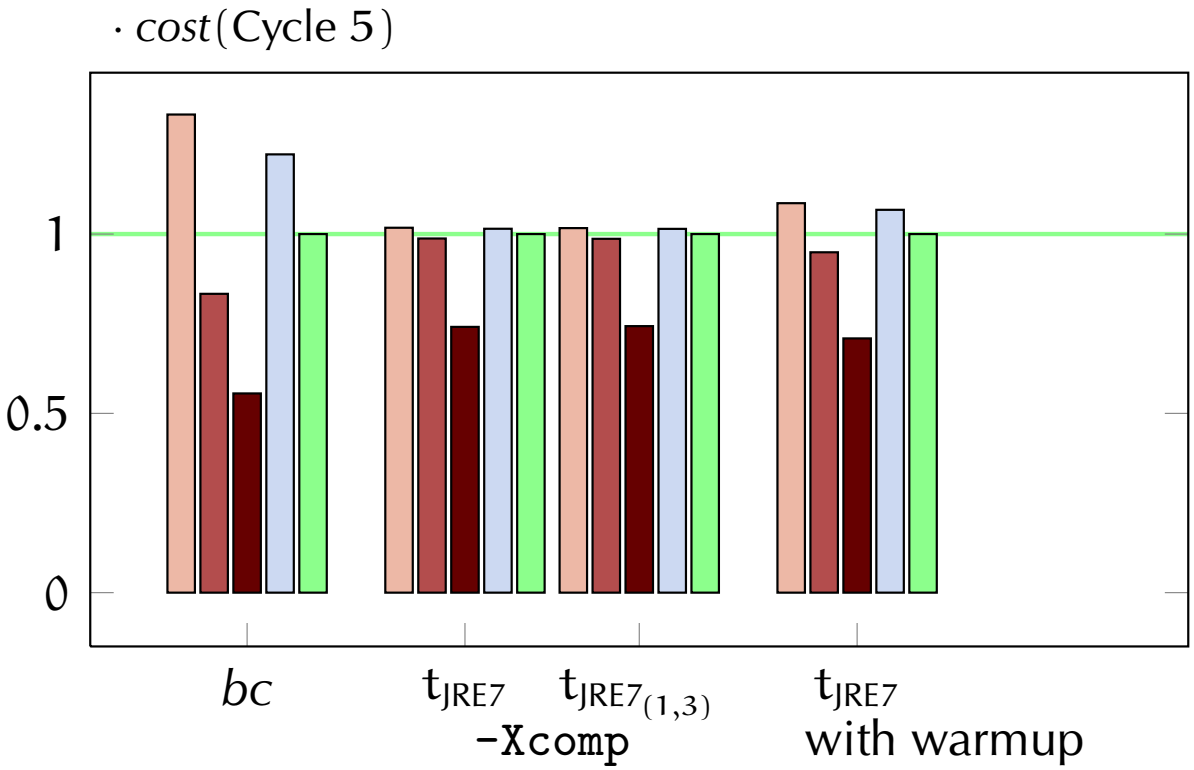
. . . stay tuned

Cycle Costs


· cost(Cycle 5)
1
0.5
0
bc
-Xcomp
with warmup

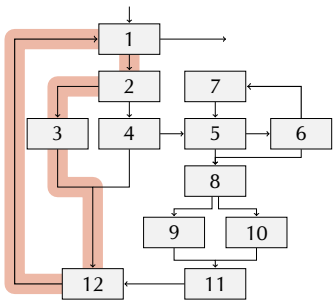

Cycle 1
Cycle 2
Cycle 3
Cycle 4
Cycle 5

Cycle Costs

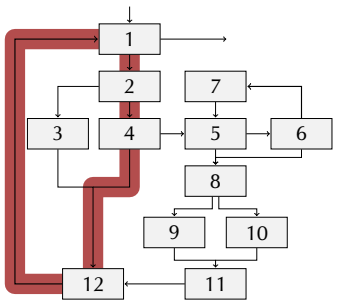


- measures **agree** qualitatively
- **but:** **smaller** difference

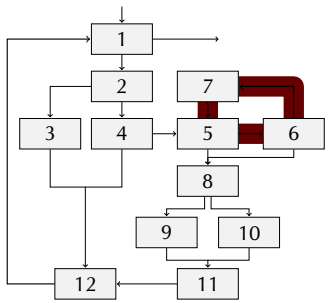
Cycle 1



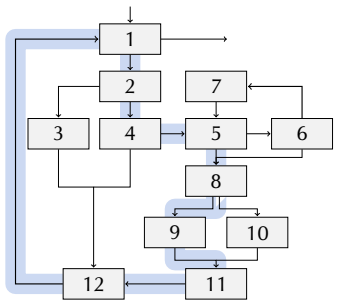
Cycle 2



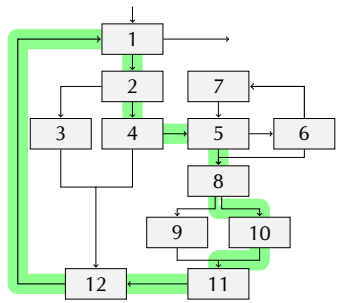
Cycle 3



Cycle 4

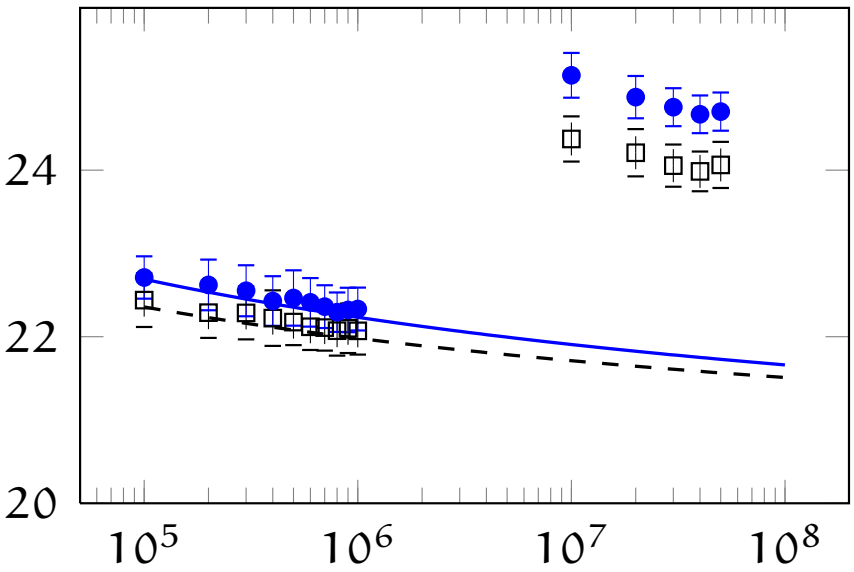


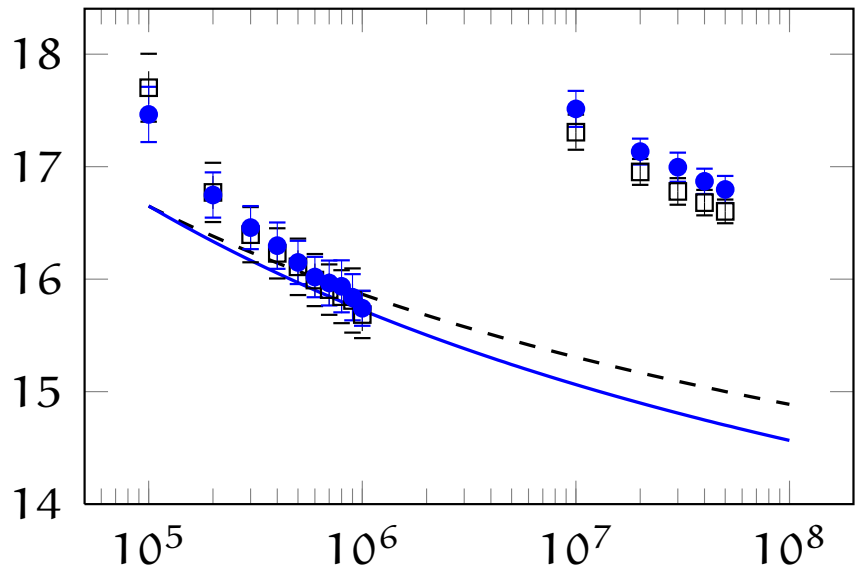
Cycle 5



Asymptotic Expected Costs

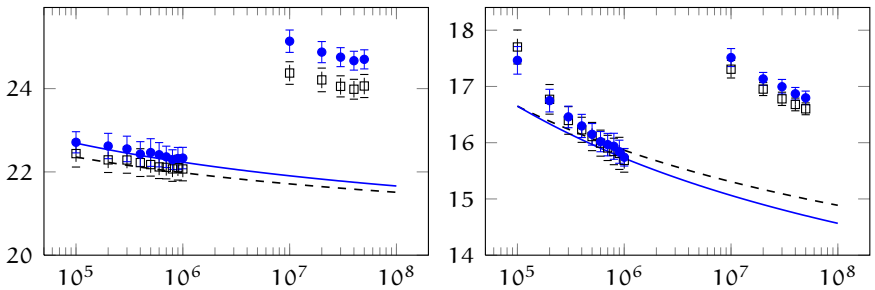
Measure		Algorithm	Random Permutations	Almost Sorted Lists
Bytecodes	A	JRE7	$19.40\,n \ln n + 51\,n$	$15.10\,n \ln n + 68\,n$
		$\text{JRE7}_{(1,3)}$	$18.73\,n \ln n + 62\,n$	$13.52\,n \ln n + 85\,n$
time `-Xcomp`	B	JRE7	$20.10\,n \ln n + 26\,n$	$11.95\,n \ln n + 54\,n$
		$\text{JRE7}_{(1,3)}$	$19.95\,n \ln n + 32\,n$	$11.09\,n \ln n + 64\,n$
time warmup	B	JRE7		
		$\text{JRE7}_{(1,3)}$		





Asymptotic Expected Costs

Measure		Algorithm	Random Permutations	Almost Sorted Lists
Bytecodes	A	JRE7	$19.40\,n\ln n + 51\,n$	$15.10\,n\ln n + 68\,n$
		$\text{JRE7}_{(1,3)}$	$18.73\,n\ln n + 62\,n$	$13.52\,n\ln n + 85\,n$
time `-Xcomp`	B	JRE7	$20.10\,n\ln n + 26\,n$	$11.95\,n\ln n + 54\,n$
		$\text{JRE7}_{(1,3)}$	$19.95\,n\ln n + 32\,n$	$11.09\,n\ln n + 64\,n$
time warmup	B	JRE7		
		$\text{JRE7}_{(1,3)}$		

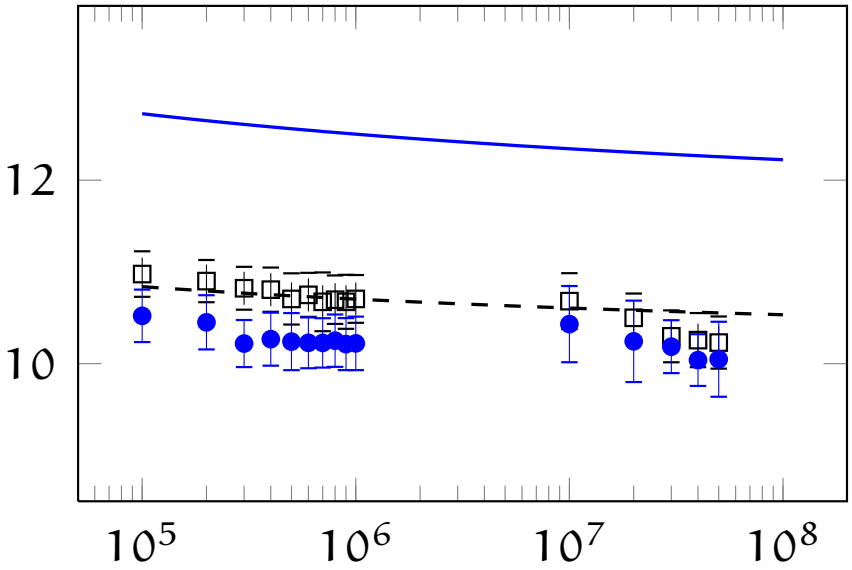


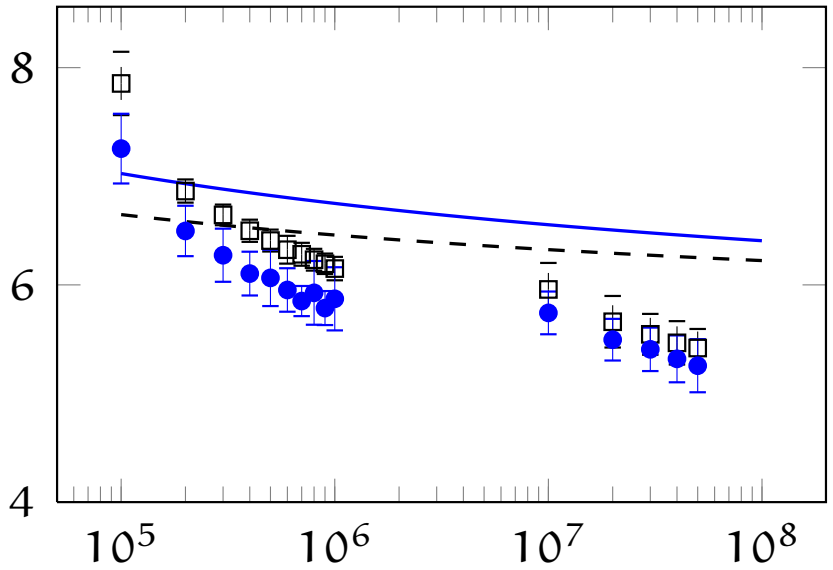
- JIT **without** profiling

$\rightsquigarrow$ asymptotically, $\text{JRE7}_{(1,3)}$ **faster**!

Asymptotic Expected Costs

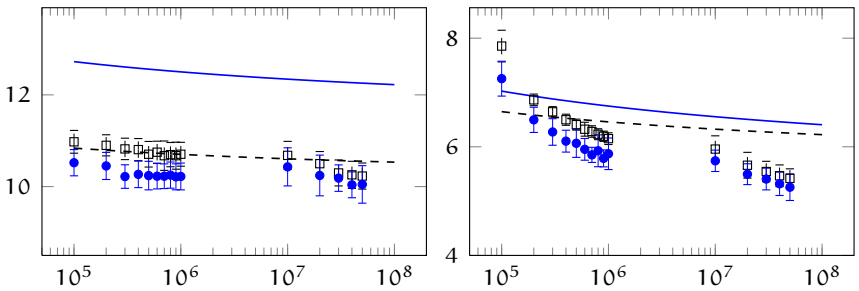
Measure		Algorithm	Random Permutations	Almost Sorted Lists
Bytecodes	A	JRE7	$19.40\,n\ln n + 51\,n$	$15.10\,n\ln n + 68\,n$
		$\text{JRE7}_{(1,3)}$	$18.73\,n\ln n + 62\,n$	$13.52\,n\ln n + 85\,n$
time -Xcomp	B	JRE7	$20.10\,n\ln n + 26\,n$	$11.95\,n\ln n + 54\,n$
		$\text{JRE7}_{(1,3)}$	$19.95\,n\ln n + 32\,n$	$11.09\,n\ln n + 64\,n$
time warmup	B	JRE7	$10.02\,n\ln n + 9\,n$	$5.52\,n\ln n + 13\,n$
		$\text{JRE7}_{(1,3)}$	$11.39\,n\ln n + 15\,n$	$5.38\,n\ln n + 19\,n$





Asymptotic Expected Costs

Measure		Algorithm	Random Permutations	Almost Sorted Lists
Bytecodes	A	□–□ JRE7	$19.40\,n\ln n + 51\,n$	$15.10\,n\ln n + 68\,n$
		●–● $\text{JRE7}_{(1,3)}$	$18.73\,n\ln n + 62\,n$	$13.52\,n\ln n + 85\,n$
time `-Xcomp`	B	□–□ JRE7	$20.10\,n\ln n + 26\,n$	$11.95\,n\ln n + 54\,n$
		●–● $\text{JRE7}_{(1,3)}$	$19.95\,n\ln n + 32\,n$	$11.09\,n\ln n + 64\,n$
time warmup	B	□–□ JRE7	$10.02\,n\ln n + 9\,n$	$5.52\,n\ln n + 13\,n$
		●–● $\text{JRE7}_{(1,3)}$	$11.39\,n\ln n + 15\,n$	$5.38\,n\ln n + 19\,n$

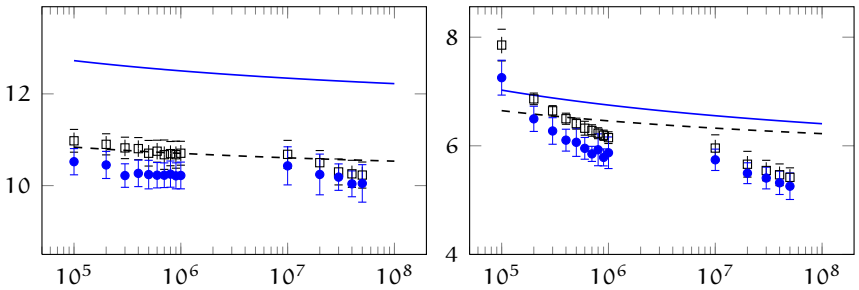


- JIT **with** profiling and warmup

⇝ asymptotically, $\text{JRE7}_{(1,3)}$ **slower**!

Asymptotic Expected Costs

Measure		Algorithm	Random Permutations	Almost Sorted Lists
Bytecodes	A	JRE7	$19.40\,n\ln n + 51\,n$	$15.10\,n\ln n + 68\,n$
		$\text{JRE7}_{(1,3)}$	$18.73\,n\ln n + 62\,n$	$13.52\,n\ln n + 85\,n$
time -Xcomp	B	JRE7	$20.10\,n\ln n + 26\,n$	$11.95\,n\ln n + 54\,n$
		$\text{JRE7}_{(1,3)}$	$19.95\,n\ln n + 32\,n$	$11.09\,n\ln n + 64\,n$
time warmup	B	JRE7	$10.02\,n\ln n + 9\,n$	$5.52\,n\ln n + 13\,n$
		$\text{JRE7}_{(1,3)}$	$11.39\,n\ln n + 15\,n$	$5.38\,n\ln n + 19\,n$

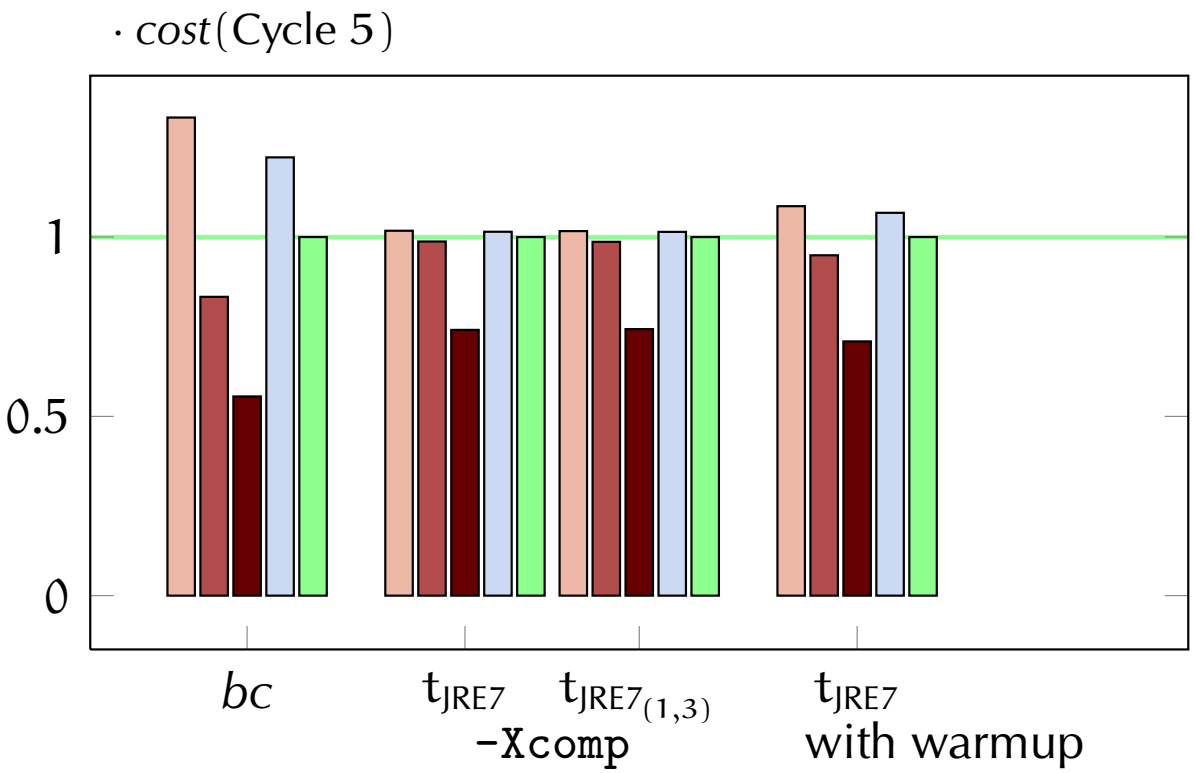


- JIT **with** profiling and warmup

$\rightsquigarrow$ asymptotically, $\text{JRE7}_{(1,3)}$ **slower**!

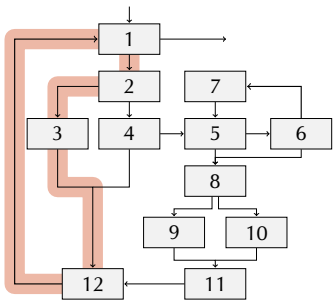
*What changes with **profiling** enabled?*

Cycle Costs

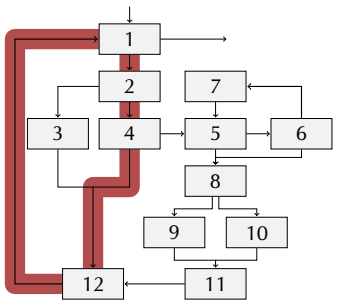


- measures **agree** qualitatively

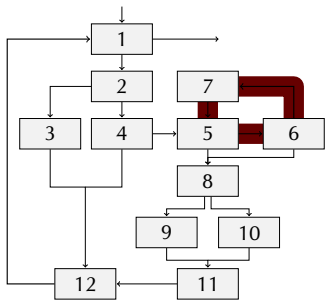
Cycle 1



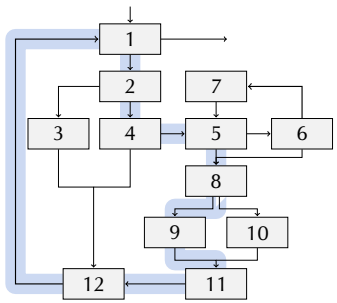
Cycle 2



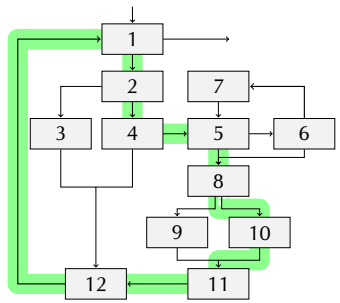
Cycle 3



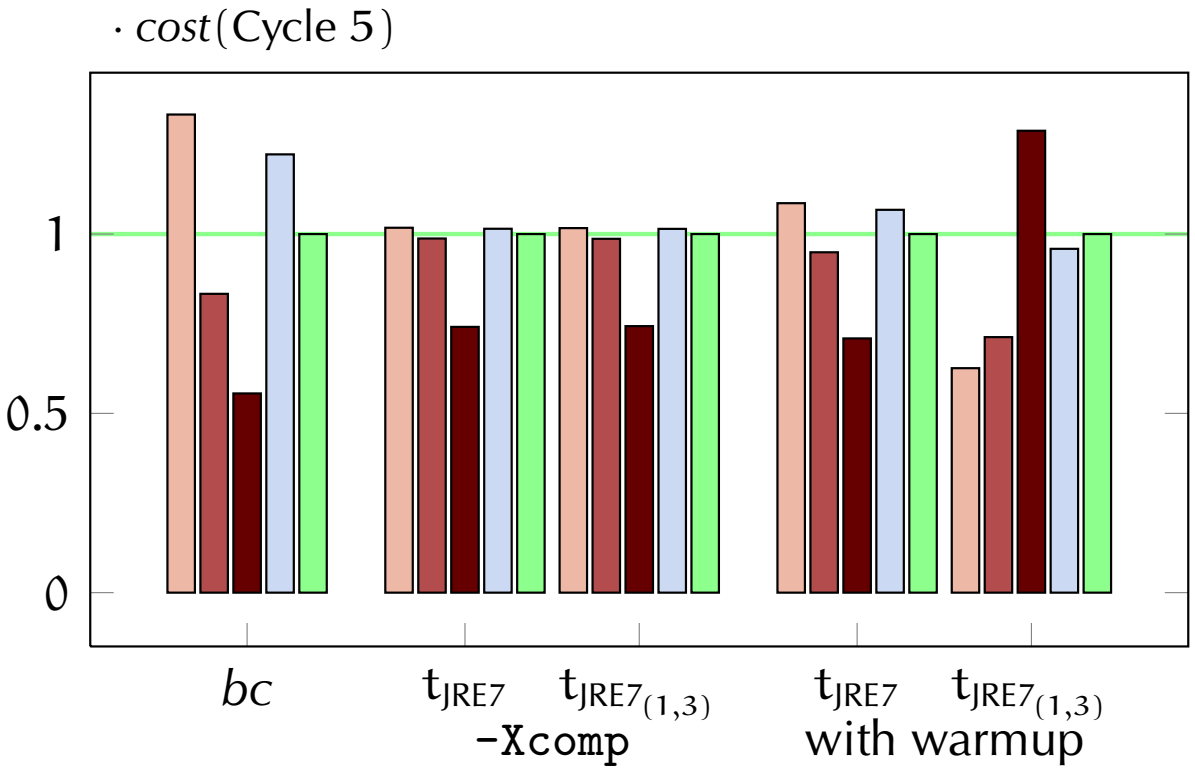
Cycle 4



Cycle 5

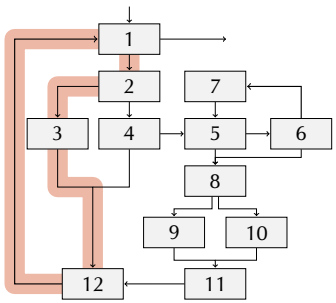


Cycle Costs

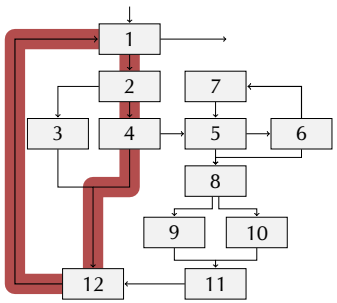


- measures **agree** qualitatively
- **except** for $\text{JRE7}_{(1,3)}$ with **profiling JIT**!

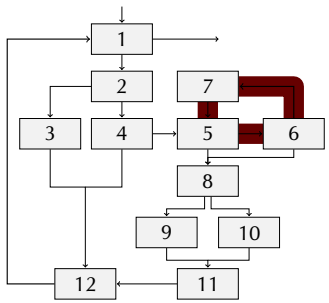
Cycle 1



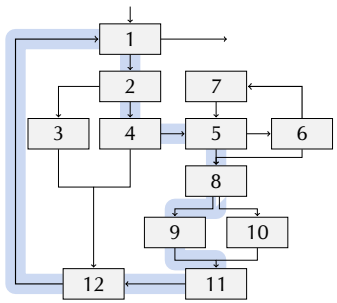
Cycle 2



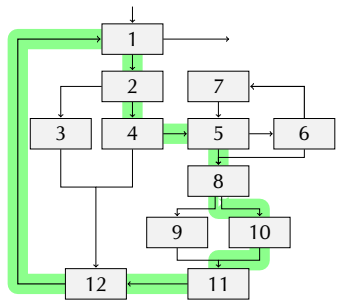
Cycle 3



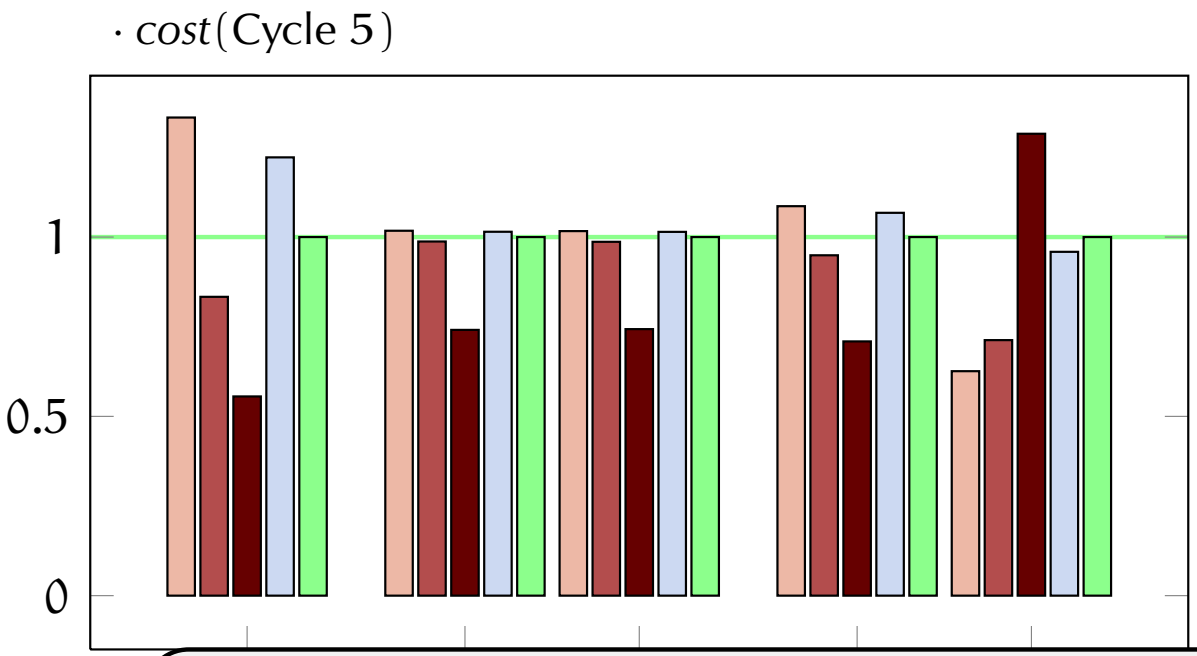
Cycle 4



Cycle 5



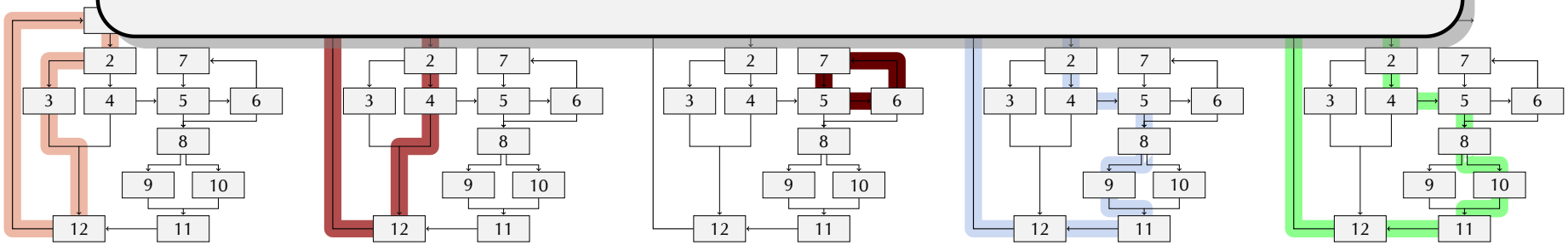
Cycle Costs



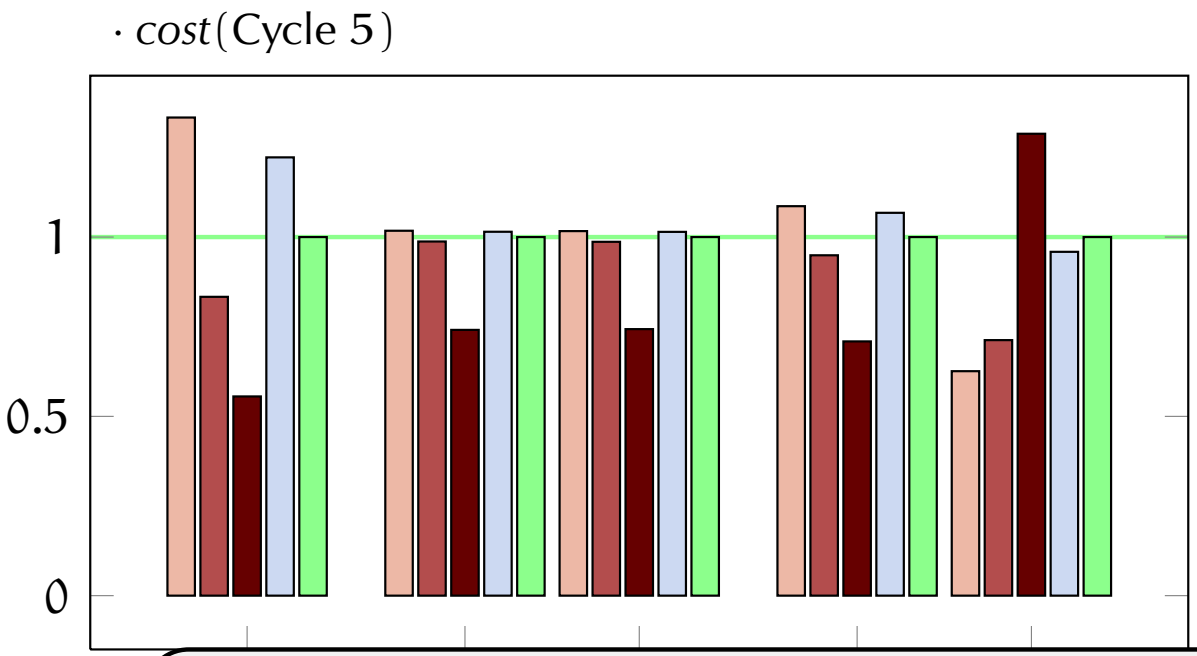
- measures **agree** qualitatively
- **except** for JRE7$_{(1,3)}$ with **profiling JIT**!

$\rightsquigarrow$ For JRE7$_{(1,3)}$, the code created by **profiling JIT** for Cycle 3 is much **slower** than for JRE7!

$\rightsquigarrow$ That's the place to focus future research on.



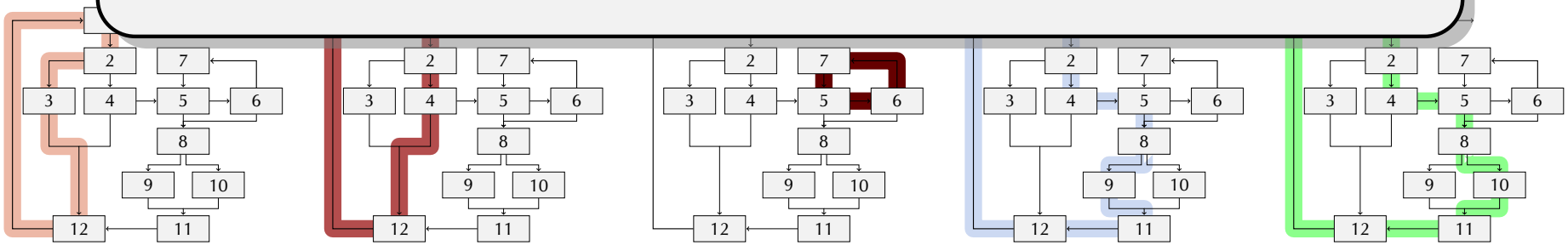
Cycle Costs



- measures **agree** qualitatively
- **except** for JRE7$_{(1,3)}$ with **profiling JIT**!

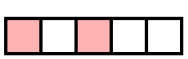
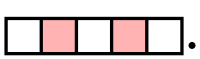
$\rightsquigarrow$ For JRE7$_{(1,3)}$, the code created by **profiling JIT** for Cycle 3 is much **slower** than for JRE7!

$\rightsquigarrow$ That's the place to focus future research on.



Conclusion

Summary

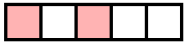
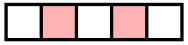
- Java 7's dual pivot Quicksort is highly **asymmetric**.
-  executes **less Bytecodes** than .
- **Almost sorted** inputs amplify impact of pivot sampling.
- Oracle's **profiling JIT** compiler creates different code for $\text{JRE7}_{(1,3)}$, which potentially overcompensates gains.
- **Control flow graph decomposition** supported by MaLiJAn makes difference in code efficiency directly visible.

Open Problems

1. What causes **different costs** for Cycle 3?
2. Are the differences idiosyncracies of **Java**/Oracle's JRE?
3. Performance of $\text{JRE7}_{(1,3)}$ on other inputs, especially with **equal keys**?

Conclusion

Summary

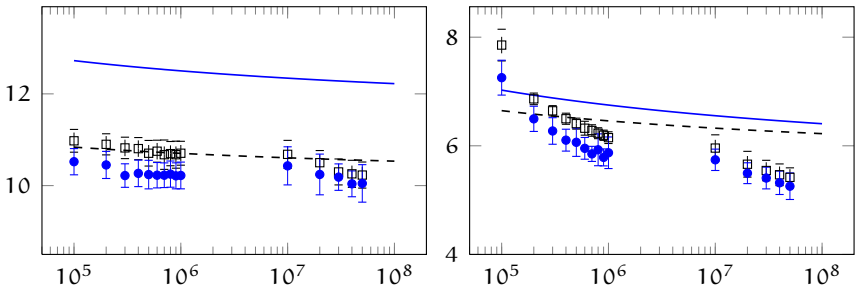
- Java 7's dual pivot Quicksort is highly **asymmetric**.
-  executes **less Bytecodes** than .
- **Almost sorted** inputs amplify impact of pivot sampling.
- Oracle's **profiling JIT** compiler creates different code for JRE7$_{(1,3)}$, which potentially overcompensates gains.
- **Control flow graph decomposition** supported by MaLiJAn makes difference in code efficiency directly visible.

Open Problems

- What causes **different costs** for Cycle 3?
- Are the differences idiosyncracies of **Java**/Oracle's JRE?
- Performance of JRE7$_{(1,3)}$ on other inputs, especially with **equal keys**?

Asymptotic Expected Costs

Measure		Algorithm	Random Permutations	Almost Sorted Lists
Bytecodes	A	JRE7	$19.40\,n\ln n + 51\,n$	$15.10\,n\ln n + 68\,n$
		$\text{JRE7}_{(1,3)}$	$18.73\,n\ln n + 62\,n$	$13.52\,n\ln n + 85\,n$
time -Xcomp	B	JRE7	$20.10\,n\ln n + 26\,n$	$11.95\,n\ln n + 54\,n$
		$\text{JRE7}_{(1,3)}$	$19.95\,n\ln n + 32\,n$	$11.09\,n\ln n + 64\,n$
time warmup	B	JRE7	$10.02\,n\ln n + 9\,n$	$5.52\,n\ln n + 13\,n$
		$\text{JRE7}_{(1,3)}$	$11.39\,n\ln n + 15\,n$	$5.38\,n\ln n + 19\,n$



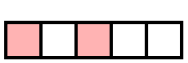
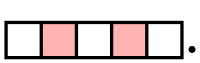
- JIT **with** profiling and warmup

$\rightsquigarrow$ asymptotically, $\text{JRE7}_{(1,3)}$ **slower**!

*What changes with **profiling** enabled?*

Conclusion

Summary

- Java 7's dual pivot Quicksort is highly **asymmetric**.
-  executes **less Bytecodes** than .
- **Almost sorted** inputs amplify impact of pivot sampling.
- Oracle's **profiling JIT** compiler creates different code for JRE7$_{(1,3)}$, which potentially overcompensates gains.
- **Control flow graph decomposition** supported by MaLiJAn makes difference in code efficiency directly visible.

Open Problems

- What causes **different costs** for Cycle 3?
- Are the differences idiosyncracies of **Java**/Oracle's JRE?
- Performance of JRE7$_{(1,3)}$ on other inputs, especially with **equal keys**?