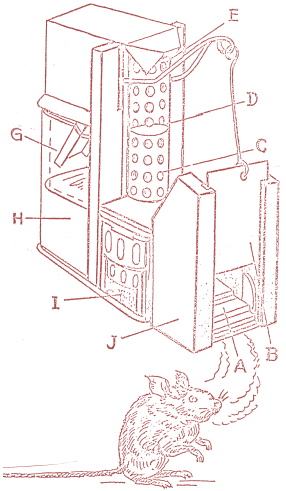
Dual-Pivot Quicksort and Beyond



Sebastian Wild

TECHNISCHE UNIVERSITÄT
KAISERSLAUTERN

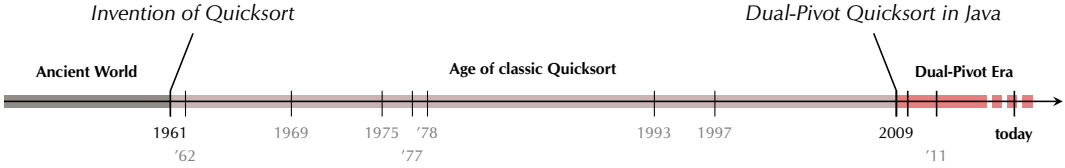
Forschungsseminar
Hasso-Plattner-Institut

6. September 2016

It is tempting to try to develop ways to improve quicksort:
a faster sorting algorithm is computer science's "better mousetrap,"
and quicksort is a venerable method that seems to invite tinkering.

R. Sedgewick and K. Wayne, Algorithms 4th ed.

Sorting History



Sorting History

1961,62 **Hoare:** publication, first analysis

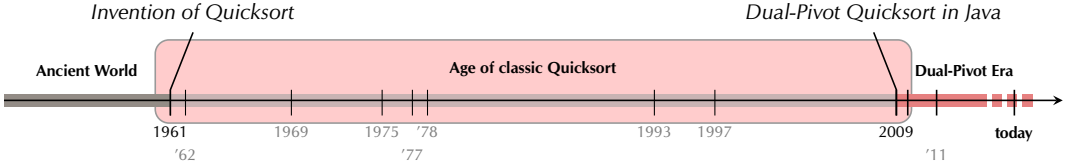
1969 **Singleton:** median-of-three & Insertionsort on small subarrays

1975-78 **Sedgewick:** analysis of many optimizations

1993 **Bentley, McIlroy:** duplicate elements & "ninther"

1997 **Musser:** $\mathcal{O}(n \log n)$ worst case by truncating recursion

$\rightsquigarrow$ Basic algorithm settled since 1961; latest tweaks from 1990's.



Sorting History

1961,62 **Hoare:** publication, first analysis

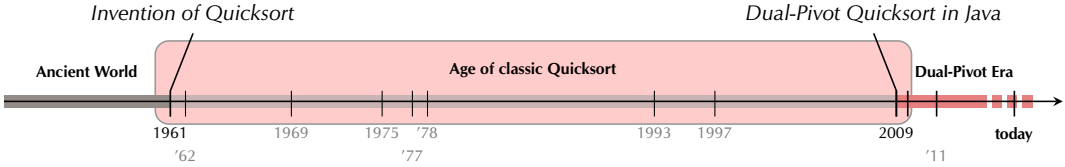
1969 **Singleton:** median-of-three & Insertionsort on small subarrays

1975-78 **Sedgewick:** analysis of many optimizations

1993 **Bentley, McIlroy:** duplicate elements & "ninther"

1997 **Musser:** $\mathcal{O}(n \log n)$ worst case by truncating recursion

$\rightsquigarrow$ Basic algorithm settled since 1961; latest tweaks from 1990's.



Sorting History

1961,62 **Hoare:** publication, first analysis

1969 **Singleton:** median-of-three & Insertionsort on small subarrays

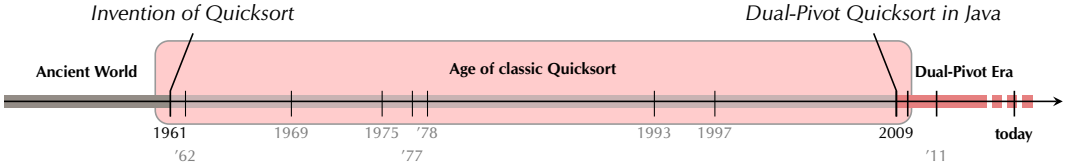
1975-78 **Sedgewick:** analysis of many optimizations

1993 **Bentley, McIlroy:** duplicate elements & "ninther"

1997 **Musser:** $\mathcal{O}(n \log n)$ worst case by truncating recursion

$\rightsquigarrow$ Basic algorithm settled since 1961; latest tweaks from 1990's.

Since then: Almost identical in all programming libraries!



Sorting History

2008 – 2009 **Vladimir Yaroslavskiy** (developer at Sun)
experiments with Quicksort with **two pivots**

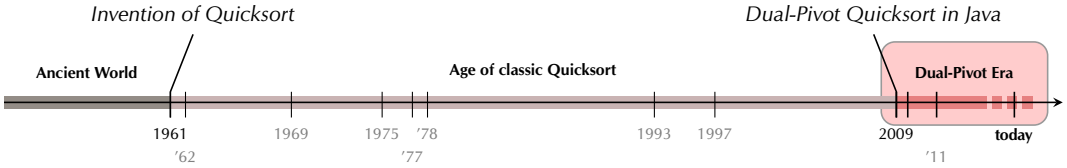
2009 – 2011 optimizations by Joshua Bloch, Jon Bentley and others

11 Sep 2009 announcement on Java core library mailing list

29 Oct 2009 **inclusion** in development version of OpenJDK

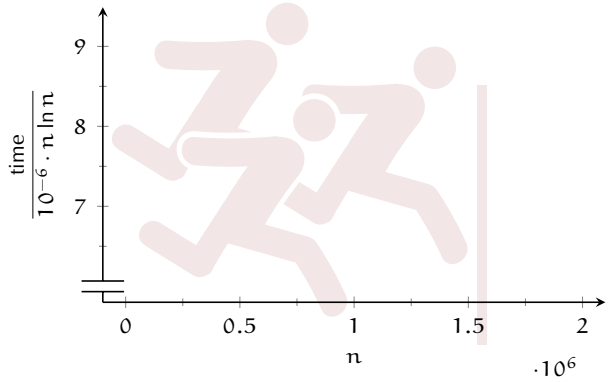
28 July 2011 **public release** of Java 7 with ***Yaroslavskiy-Bentley-Block (YBB) Quicksort***

(same in Java 8)



Running Time Experiments

Why change a well-tested implementation?



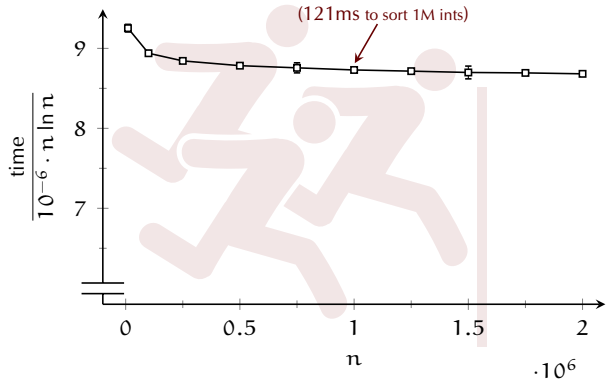
Normalized Java runtimes (in *ms*).

Average of 1000 random permutations per size.

- YBB Quicksort significantly faster!
- . . . just new engineering tricks?

Running Time Experiments

Why change a well-tested implementation?



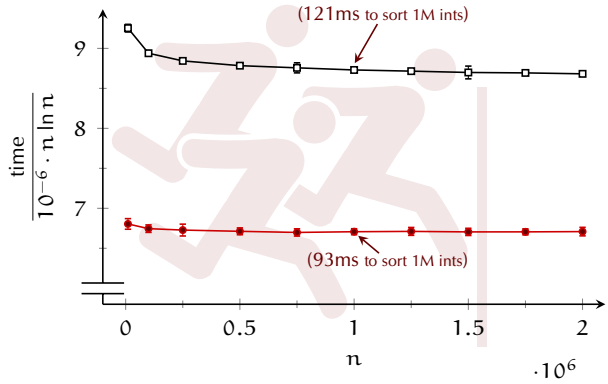
Normalized Java runtimes (in *ms*).

Average of 1000 random permutations per size.

- YBB Quicksort significantly faster!
- . . . just new engineering tricks?

Running Time Experiments

Why change a well-tested implementation?



Java 6 Library
Java 7 Library

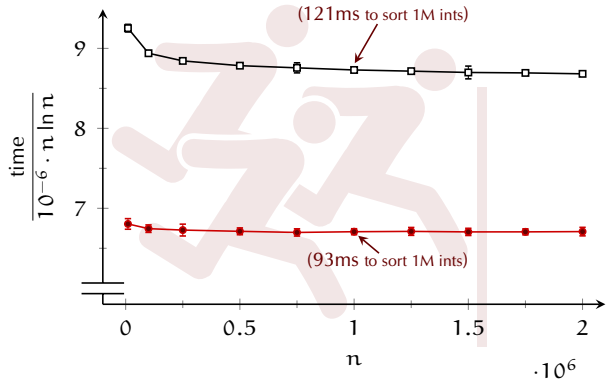
Normalized Java runtimes (in *ms*).

Average of 1000 random permutations per size.

- YBB Quicksort significantly faster!
- . . . just new engineering tricks?

Running Time Experiments

Why change a well-tested implementation?



Java 6 Library
Java 7 Library

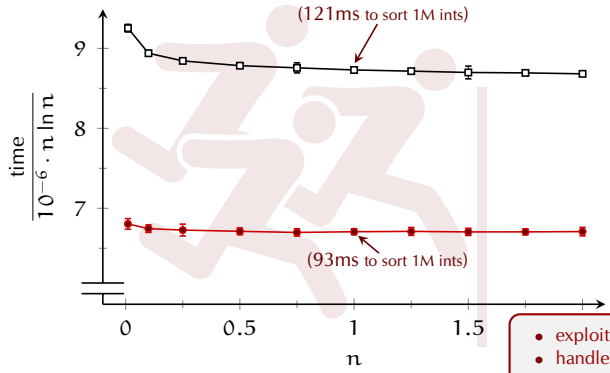
Normalized Java runtimes (in *ms*).

Average of 1000 random permutations per size.

- YBB Quicksort significantly faster!
- . . . just new engineering tricks?

Running Time Experiments

Why change a well-tested implementation?



Normalized Java runtimes (in *ms*).

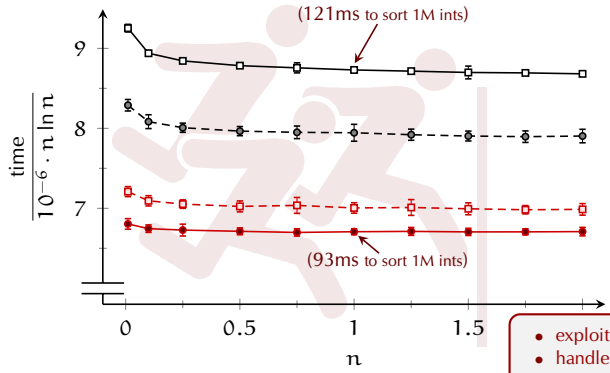
Average of 1000 random permutations per size.

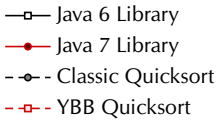
- YBB Quicksort significantly faster!
- . . . just new engineering tricks?

- exploit (partially) sorted inputs
- handle equal items efficiently
- Insertionsort for small subproblems (actually pair insertion)
- choose good pivots
- use sentinel elements
- low-level optimizations

Running Time Experiments

Why change a well-tested implementation?





Normalized Java runtimes (in *ms*).

Average of 1000 random permutations per size.

- YBB Quicksort significantly faster!
- . . . just new engineering tricks?

- exploit (partially) sorted inputs
- handle equal items efficiently
- Insertionsort for small subproblems (actually pair insertion)
- choose good pivots
- use sentinel elements
- low-level optimizations

- **No:** remains true for **basic** variants of algorithms: -●- vs. -□-!

Dual-Pivot Quicksort: Conceptual View

Single-Pivot Quicksort

1. Choose pivots P.
2. For each element x, determine its **class**
   - small for $x < P$
   - large for $P < x$

   by comparing x to P.
3. Arrange elements according to classes:

4. Sort subarrays recursively.

Dual-Pivot Quicksort: Conceptual View

Single-Pivot Quicksort

1. Choose pivots P.
2. For each element x, determine its **class**
   - small for $x < P$
   - large for $P < x$

   by comparing x to P.
3. Arrange elements according to classes:

4. Sort subarrays recursively.

Dual-Pivot Quicksort: Conceptual View

Single-Pivot Quicksort

1. Choose pivots P.
2. For each element x, determine its **class**
   - **small** for $x < P$
   - **large** for $P < x$

   by comparing x to P.
3. Arrange elements according to classes:

4. Sort subarrays recursively.

Dual-Pivot Quicksort: Conceptual View

Single-Pivot Quicksort

1. Choose pivots P.
2. For each element x, determine its **class**
   - **small** for $x < P$
   - **large** for $P < x$

   assuming distinct elements in this talk

   by comparing x to P.
3. Arrange elements according to classes:

4. Sort subarrays recursively.

Dual-Pivot Quicksort: Conceptual View

Single-Pivot Quicksort

1. Choose pivots P.
2. For each element x, determine its **class**
   - **small** for $x < P$
   - **large** for $P < x$

   assuming distinct elements in this talk

   by comparing x to P.
3. Arrange elements according to classes:



4. Sort subarrays recursively.

Dual-Pivot Quicksort: Conceptual View

Single-Pivot Quicksort

1. Choose pivots P.
2. For each element x, determine its **class**
   - **small** for $x < P$
   - **large** for $P < x$

   assuming distinct elements in this talk

   by comparing x to P.
3. Arrange elements according to classes:



4. Sort subarrays recursively.

Dual-Pivot Quicksort: Conceptual View

Single-Pivot Quicksort

1. Choose pivots P.
2. For each element x, determine its **class**
   - **small** for $x < P$
   - **large** for $P < x$

   assuming distinct elements in this talk

   by comparing x to P.
3. Arrange elements according to classes:



4. Sort subarrays recursively.

Dual-Pivot Quicksort

1. Choose **two pivots** $P \leqslant Q$
2. For each element x, determine its **class**
   - **small** for $x < P$
   - **medium** for $P < x < Q$
   - **large** for $Q < x$

   by comparing x to **pivots** P and Q
3. Arrange elements according to classes:

4. Sort subarrays recursively.

Dual-Pivot Quicksort: Conceptual View

Single-Pivot Quicksort

1. Choose pivots P.
2. For each element x, determine its **class**
   - **small** for $x < P$
   - **large** for $P < x$

   assuming distinct elements in this talk

   by comparing x to P.
3. Arrange elements according to classes:



4. Sort subarrays recursively.

Dual-Pivot Quicksort

1. Choose **two pivots** $P \leqslant Q$
2. For each element x, determine its **class**
   - **small** for $x < P$
   - **medium** for $P < x < Q$
   - **large** for $Q < x$

   by comparing x to **pivots** P and Q
3. Arrange elements according to classes:

4. Sort subarrays recursively.

Dual-Pivot Quicksort: Conceptual View

Single-Pivot Quicksort

1. Choose pivots P.
2. For each element x, determine its **class**
   - **small** for $x < P$
   - **large** for $P < x$

   assuming distinct elements in this talk

   by comparing x to P.
3. Arrange elements according to classes:



4. Sort subarrays recursively.

Dual-Pivot Quicksort

1. Choose **two pivots** $P \leqslant Q$
2. For each element x, determine its **class**
   - **small** for $x < P$
   - **medium** for $P < x < Q$
   - **large** for $Q < x$

   by comparing x to **pivots** P and Q
3. Arrange elements according to classes:

4. Sort subarrays recursively.

Dual-Pivot Quicksort: Conceptual View

Single-Pivot Quicksort

1. Choose pivots P.
2. For each element x, determine its **class**
   - **small** for $x < P$
   - **large** for $P < x$

   assuming distinct elements in this talk

   by comparing x to P.
3. Arrange elements according to classes:



4. Sort subarrays recursively.

Dual-Pivot Quicksort

1. Choose **two pivots** $P \leqslant Q$
2. For each element x, determine its **class**
   - **small** for $x < P$
   - **medium** for $P < x < Q$
   - **large** for $Q < x$

   by comparing x to **pivots** P and Q
3. Arrange elements according to classes:



4. Sort subarrays recursively.

Dijkstra's Dutch National Flag Problem

Dual-Pivot Quicksort: Conceptual View

Single-Pivot Quicksort

1. Choose pivots P.
2. For each element x, determine its **class**
   - **small** for $x < P$
   - **large** for $P < x$

   assuming distinct elements in this talk

   by comparing x to P.
3. Arrange elements according to classes:



4. Sort subarrays recursively.

Dijkstra's Dutch National Flag Problem

Dual-Pivot Quicksort

1. Choose **two pivots** $P \leqslant Q$
2. For each element x, determine its **class**
   - **small** for $x < P$
   - **medium** for $P < x < Q$
   - **large** for $Q < x$

   by comparing x to **pivots** P and Q
3. Arrange elements according to classes:



4. Sort subarrays recursively.

Dual-Pivot Quicksort: Conceptual View

Single-Pivot Quicksort

1. Choose pi
2. For each element x, determine its **class**
   - **small** for $x < P$
   - **large** for $P < x$

   assuming distinct elements in this talk

   by comparing x to P.
3. Arrange elements according to classes:



4. Sort subarrays recursively.

Dual-Pivot Quicksort

2. For each element x, determine its **class**
   - **small** for $x < P$
   - **medium** for $P < x < Q$
   - **large** for $Q < x$

   by comparing x to **pivots** P and Q

Dijkstra's Dutch National Flag Problem

3. Arrange elements according to classes:



4. Sort subarrays recursively.

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

Dual-Pivot Quicksort: Conceptual View

Single-Pivot Quicksort

1. Choose pi
2. For each element x, determine its **class**
   - **small** for $x < P$
   - **large** for $P < x$

   assuming distinct elements in this talk

   by comparing x to P.
3. Arrange elements according to classes:



4. Sort subarrays recursively.

Dual-Pivot Quicksort

2. For each element x, determine its **class**
   - **small** for $x < P$
   - **medium** for $P < x < Q$
   - **large** for $Q < x$

   by comparing x to **pivots** P and Q

Dijkstra's Dutch National Flag Problem

3. Arrange elements according to classes:



4. Sort subarrays recursively.

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

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 )
```

YBBQuicksort(A, *left*, *right*)

```
1  if right ⩽ left then return
2  P = A[left];   Q = A[right]
3  if P > Q then Swap P and Q end if
4  ℓ = left + 1;   g = right − 1;   k = ℓ
5  while k ⩽ g
6      if A[k] < P
7          Swap A[k] and A[ℓ] ;  ℓ = ℓ + 1
8      else if A[k] ⩾ Q
9          while A[g] > Q and k < g
10             g = g − 1
11         end while
12         Swap A[k] and A[g] ;  g = g − 1
13         if A[k] < P
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 )
```

YBBQuicksort(A, *left*, *right*)

```
1  if right ⩽ left then return
2  P = A[left];   Q = A[right]
3  if P > Q then Swap P and Q end if
4  ℓ = left + 1;   g = right − 1;   k = ℓ
5  while k ⩽ g
6      if A[k] < P
7          Swap A[k] and A[ℓ] ; ℓ = ℓ + 1
8      else if A[k] ⩾ Q
9          while A[g] > Q and k < g
10             g = g − 1
11         end while
12         Swap A[k] and A[g] ;  g = g − 1
13         if A[k] < P
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
- **YBB Quicksort**
  - ...looks messy

YBBQuicksort(A, *left*, *right*)

```
1  if right ⩽ left then return
2  P = A[left];   Q = A[right]
3  if P > Q then Swap P and Q end if
4  ℓ = left + 1;   g = right − 1;   k = ℓ
5  while k ⩽ g
6      if A[k] < P
7          Swap A[k] and A[ℓ] ; ℓ = ℓ + 1
8      else if A[k] ⩾ Q
9          while A[g] > Q and k < g
10             g = g − 1
11         end while
12         Swap A[k] and A[g] ;  g = g − 1
13         if A[k] < P
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
- **YBB Quicksort**
  - ...looks messy

YBBQuicksort(A, *left*, *right*)

```
1  if right ⩽ left then return
2  P = A[left];   Q = A[right]
3  if P > Q then Swap P and Q end if
4  ℓ = left + 1;   g = right − 1;   k = ℓ
5  while k ⩽ g
6      if A[k] < P
7          Swap A[k] and A[ℓ] ; ℓ = ℓ + 1
8      else if A[k] ⩾ Q
9          while A[g] > Q and k < g
10             g = g − 1
11         end while
12         Swap A[k] and A[g] ;  g = g − 1
13         if A[k] < P
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 
- **YBB Quicksort**
  - ...looks messy

YBBQuicksort(A, *left*, *right*)

```
1  if right ⩽ left then return
2  P = A[left];   Q = A[right]
3  if P > Q then Swap P and Q end if
4  ℓ = left + 1;   g = right − 1;   k = ℓ
5  while k ⩽ g
6      if A[k] < P
7          Swap A[k] and A[ℓ] ; ℓ = ℓ + 1
8      else if A[k] ⩾ Q
9          while A[g] > Q and k < g
10             g = g − 1
11         end while
12         Swap A[k] and A[g] ;  g = g − 1
13         if A[k] < P
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 
- **YBB Quicksort**
  - . . . looks messy

YBBQuicksort(A, *left*, *right*)

```
1  if right ⩽ left then return
2  P = A[left];   Q = A[right]
3  if P > Q then Swap P and Q end if
4  ℓ = left + 1;   g = right − 1;   k = ℓ
5  while k ⩽ g
6      if A[k] < P
7          Swap A[k] and A[ℓ] ; ℓ = ℓ + 1
8      else if A[k] ⩾ Q
9          while A[g] > Q and k < g
10             g = g − 1
11         end while
12         Swap A[k] and A[g] ;  g = g − 1
13         if A[k] < P
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*)

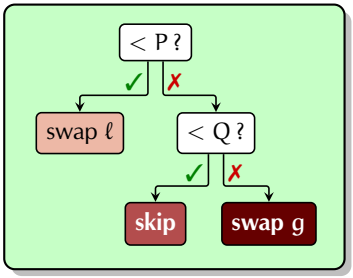
```
1  if right ⩽ left then return
2  P = A[right]
3  k = left − 1;  g = right
4  while true
5      do k = k + 1 while A[k] < P end while
6      do g = g − 1 while A[g] > P end while
7      if k ⩾ g then break while end if
8      Swap A[k] and A[g]
9  end while
10 Swap A[k] and A[right]
11 ClassicQuicksort(A,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

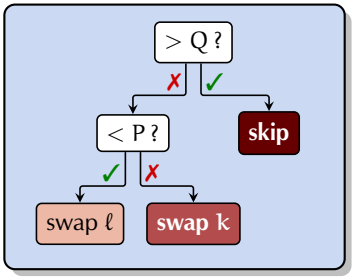
- **Classic Quicksort**
  - crossing-pointer scheme 
  - symmetric 
- **YBB Quicksort**
  - . . . looks messy 

YBBQuicksort(A, *left*, *right*)

```
1  if right ⩽ left then return
2  P = A[left];   Q = A[right]
3  if P > Q then Swap P and Q end if
4  ℓ = left + 1;   g = right − 1;   k = ℓ
5  while k ⩽ g
6      if A[k] < P
7          Swap A[k] and A[ℓ] ;  ℓ = ℓ + 1
8      else if A[k] ⩾ Q
9          while A[g] > Q and k < g
10             g = g − 1
11         end while
12         Swap A[k] and A[g] ;  g = g − 1
13         if A[k] < P
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 )
```

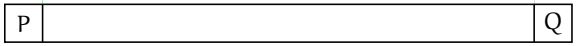
YBB Partitioning Example





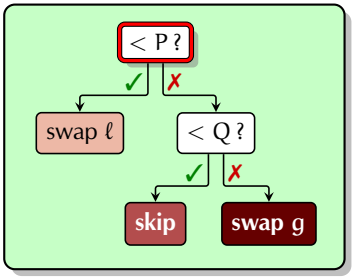


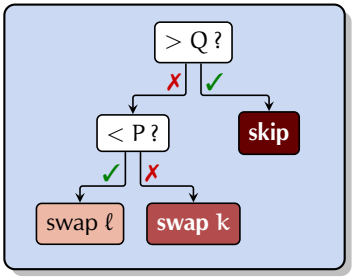
Invariant:




```
YBBQuicksort(A, left, right)
 1  if right ⩽ left then return
 2  P = A[left];   Q = A[right]
 3  if P > Q then Swap P and Q end if
 4  ℓ = left + 1;   g = right − 1;   k = ℓ
 5  while k ⩽ g
 6      if A[k] < P
 7          Swap A[k] and A[ℓ] ;  ℓ = ℓ + 1
 8      else if A[k] ⩾ Q
 9          while A[g] > Q and k < g
10              g = g − 1
11          end while
12          Swap A[k] and A[g] ;  g = g − 1
13          if A[k] < P
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  )
```

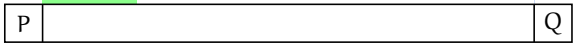
YBB Partitioning Example





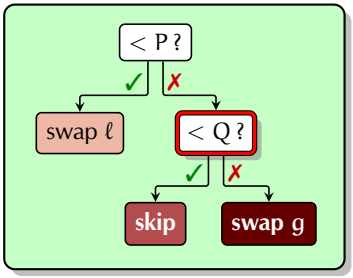


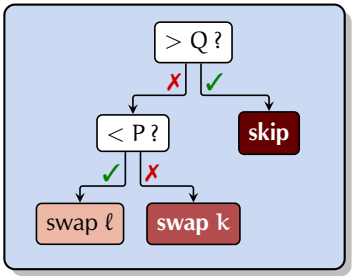
Invariant:




```
YBBQuicksort(A, left, right)
 1  if right ⩽ left then return
 2  P = A[left];   Q = A[right]
 3  if P > Q then Swap P and Q end if
 4  ℓ = left + 1;   g = right − 1;   k = ℓ
 5  while k ⩽ g
 6      if A[k] < P
 7          Swap A[k] and A[ℓ] ;  ℓ = ℓ + 1
 8      else if A[k] ⩾ Q
 9          while A[g] > Q and k < g
10              g = g − 1
11          end while
12          Swap A[k] and A[g] ;  g = g − 1
13          if A[k] < P
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  )
```

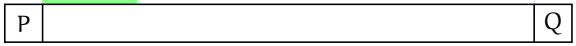
YBB Partitioning Example





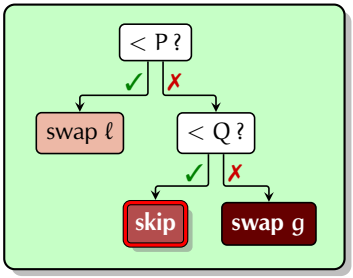


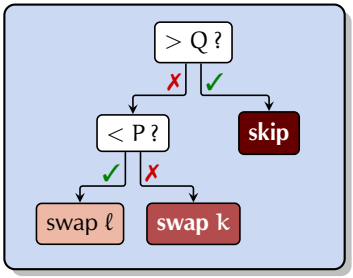
Invariant:




```
YBBQuicksort(A, left, right)
 1  if right ⩽ left then return
 2  P = A[left];   Q = A[right]
 3  if P > Q then Swap P and Q end if
 4  ℓ = left + 1;   g = right − 1;   k = ℓ
 5  while k ⩽ g
 6      if A[k] < P
 7          Swap A[k] and A[ℓ] ;  ℓ = ℓ + 1
 8      else if A[k] ⩾ Q
 9          while A[g] > Q and k < g
10              g = g − 1
11          end while
12          Swap A[k] and A[g] ;  g = g − 1
13          if A[k] < P
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  )
```

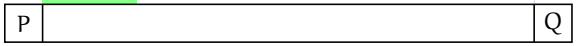
YBB Partitioning Example





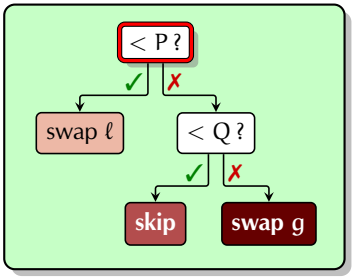


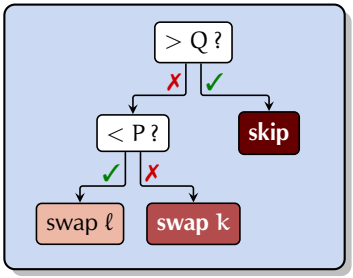
Invariant:




```
YBBQuicksort(A, left, right)
 1  if right ⩽ left then return
 2  P = A[left];   Q = A[right]
 3  if P > Q then Swap P and Q end if
 4  ℓ = left + 1;   g = right − 1;   k = ℓ
 5  while k ⩽ g
 6      if A[k] < P
 7          Swap A[k] and A[ℓ] ; ℓ = ℓ + 1
 8      else if A[k] ⩾ Q
 9          while A[g] > Q and k < g
10              g = g − 1
11          end while
12          Swap A[k] and A[g] ; g = g − 1
13          if A[k] < P
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 )
```

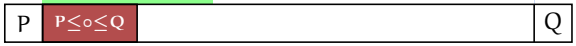
YBB Partitioning Example





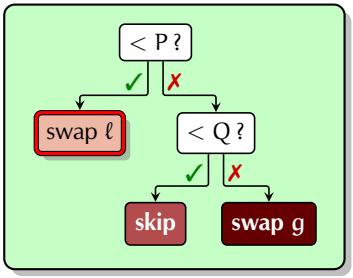


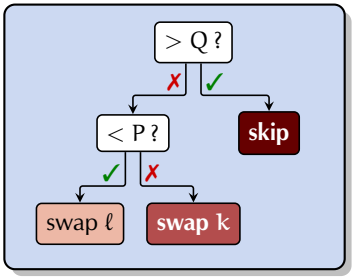
Invariant:




```
YBBQuicksort(A, left, right)
 1  if right ⩽ left then return
 2  P = A[left];   Q = A[right]
 3  if P > Q then Swap P and Q end if
 4  ℓ = left + 1;   g = right − 1;   k = ℓ
 5  while k ⩽ g
 6      if A[k] < P
 7          Swap A[k] and A[ℓ] ; ℓ = ℓ + 1
 8      else if A[k] ⩾ Q
 9          while A[g] > Q and k < g
10              g = g − 1
11          end while
12          Swap A[k] and A[g] ; g = g − 1
13          if A[k] < P
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



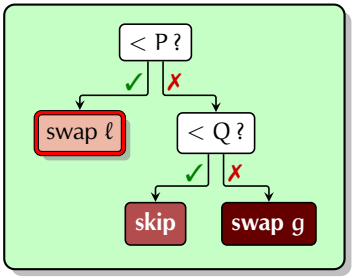


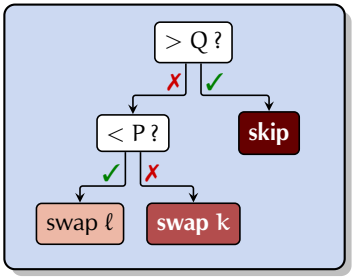


Invariant: P | P≤∘≤Q | Q

```
YBBQuicksort(A, left, right)
 1  if right ⩽ left then return
 2  P = A[left];   Q = A[right]
 3  if P > Q then Swap P and Q end if
 4  ℓ = left + 1;   g = right − 1;   k = ℓ
 5  while k ⩽ g
 6      if A[k] < P
 7          Swap A[k] and A[ℓ] ; ℓ = ℓ + 1
 8      else if A[k] ⩾ Q
 9          while A[g] > Q and k < g
10              g = g − 1
11          end while
12          Swap A[k] and A[g] ; g = g − 1
13          if A[k] < P
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 )
```

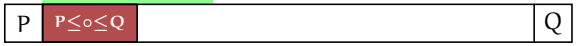
YBB Partitioning Example





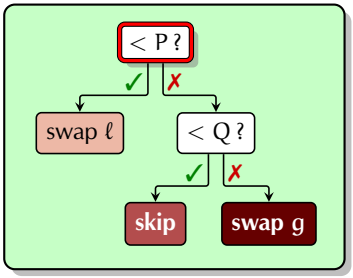


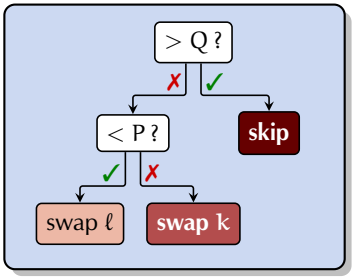
Invariant:



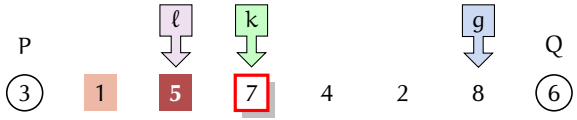

```
YBBQuicksort(A, left, right)
 1  if right ⩽ left then return
 2  P = A[left];   Q = A[right]
 3  if P > Q then Swap P and Q end if
 4  ℓ = left + 1;   g = right − 1;   k = ℓ
 5  while k ⩽ g
 6      if A[k] < P
 7          Swap A[k] and A[ℓ] ; ℓ = ℓ + 1
 8      else if A[k] ⩾ Q
 9          while A[g] > Q and k < g
10              g = g − 1
11          end while
12          Swap A[k] and A[g] ; g = g − 1
13          if A[k] < P
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

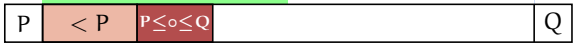




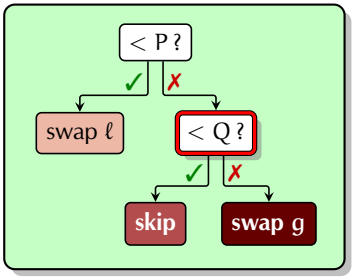

```
YBBQuicksort(A, left, right)
 1  if right ⩽ left then return
 2  P = A[left];   Q = A[right]
 3  if P > Q then Swap P and Q end if
 4  ℓ = left + 1;   g = right − 1;   k = ℓ
 5  while k ⩽ g
 6      if A[k] < P
 7          Swap A[k] and A[ℓ] ; ℓ = ℓ + 1
 8      else if A[k] ⩾ Q
 9          while A[g] > Q and k < g
10              g = g − 1
11          end while
12          Swap A[k] and A[g] ; g = g − 1
13          if A[k] < P
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 )
```

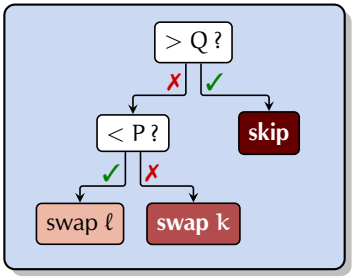


Invariant:

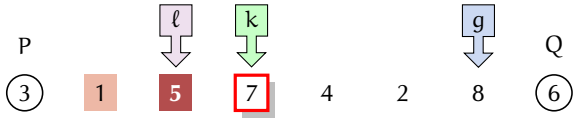


YBB Partitioning Example

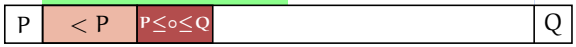




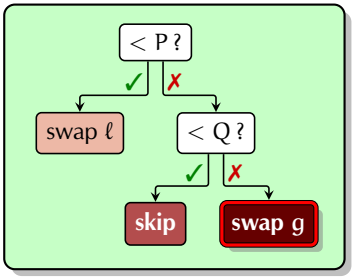

```
YBBQuicksort(A, left, right)
 1  if right ⩽ left then return
 2  P = A[left];   Q = A[right]
 3  if P > Q then Swap P and Q end if
 4  ℓ = left + 1;   g = right − 1;   k = ℓ
 5  while k ⩽ g
 6      if A[k] < P
 7          Swap A[k] and A[ℓ] ; ℓ = ℓ + 1
 8      else if A[k] ⩾ Q
 9          while A[g] > Q and k < g
10              g = g − 1
11          end while
12          Swap A[k] and A[g] ; g = g − 1
13          if A[k] < P
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 )
```

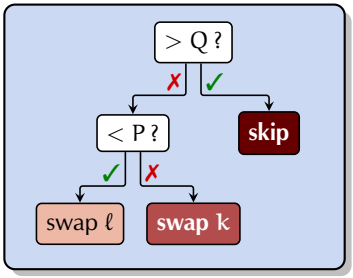


Invariant:

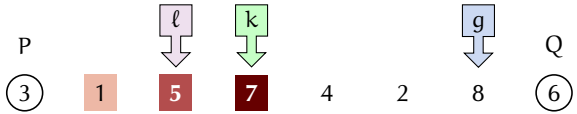


YBB Partitioning Example

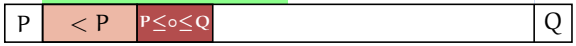




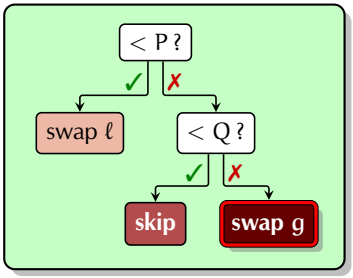

```
YBBQuicksort(A, left, right)
 1  if right ⩽ left then return
 2  P = A[left];   Q = A[right]
 3  if P > Q then Swap P and Q end if
 4  ℓ = left + 1;   g = right − 1;   k = ℓ
 5  while k ⩽ g
 6      if A[k] < P
 7          Swap A[k] and A[ℓ] ; ℓ = ℓ + 1
 8      else if A[k] ⩾ Q
 9          while A[g] > Q and k < g
10              g = g − 1
11          end while
12          Swap A[k] and A[g] ; g = g − 1
13          if A[k] < P
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 )
```

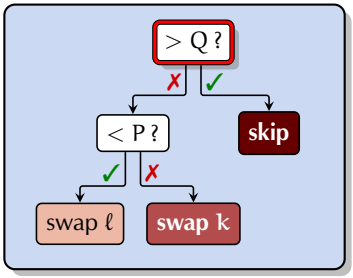


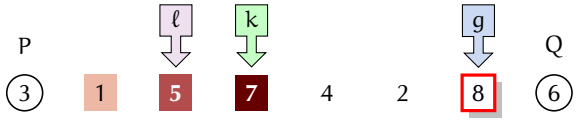
Invariant:



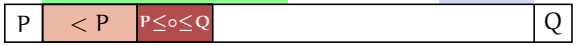
YBB Partitioning Example





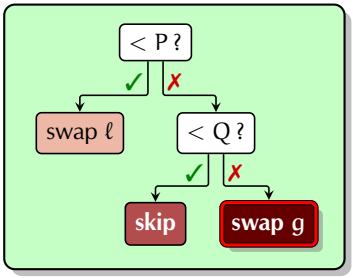


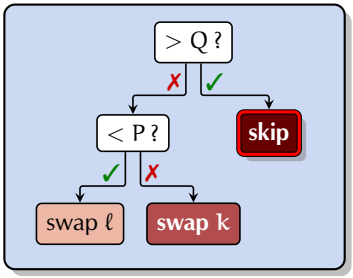
Invariant:



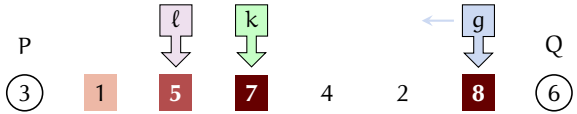

```
YBBQuicksort(A, left, right)
 1  if right ⩽ left then return
 2  P = A[left];   Q = A[right]
 3  if P > Q then Swap P and Q end if
 4  ℓ = left + 1;   g = right − 1;   k = ℓ
 5  while k ⩽ g
 6      if A[k] < P
 7          Swap A[k] and A[ℓ] ; ℓ = ℓ + 1
 8      else if A[k] ⩾ Q
 9          while A[g] > Q and k < g
10              g = g − 1
11          end while
12          Swap A[k] and A[g] ; g = g − 1
13          if A[k] < P
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

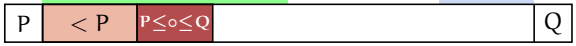




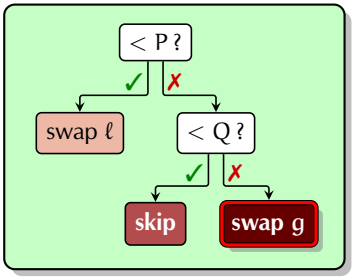

```
YBBQuicksort(A, left, right)
 1  if right ⩽ left then return
 2  P = A[left];   Q = A[right]
 3  if P > Q then Swap P and Q end if
 4  ℓ = left + 1;   g = right − 1;   k = ℓ
 5  while k ⩽ g
 6      if A[k] < P
 7          Swap A[k] and A[ℓ] ;  ℓ = ℓ + 1
 8      else if A[k] ⩾ Q
 9          while A[g] > Q and k < g
10              g = g − 1
11          end while
12          Swap A[k] and A[g] ;  g = g − 1
13          if A[k] < P
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  )
```

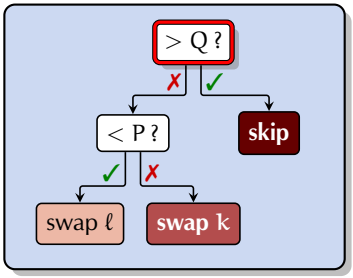


Invariant:

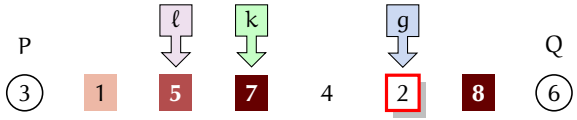


YBB Partitioning Example

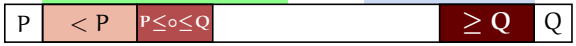




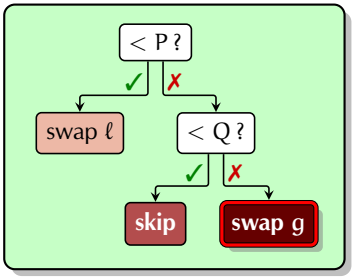

```
YBBQuicksort(A, left, right)
 1  if right ⩽ left then return
 2  P = A[left];   Q = A[right]
 3  if P > Q then Swap P and Q end if
 4  ℓ = left + 1;   g = right − 1;   k = ℓ
 5  while k ⩽ g
 6      if A[k] < P
 7          Swap A[k] and A[ℓ] ; ℓ = ℓ + 1
 8      else if A[k] ⩾ Q
 9          while A[g] > Q and k < g
10              g = g − 1
11          end while
12          Swap A[k] and A[g] ; g = g − 1
13          if A[k] < P
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 )
```

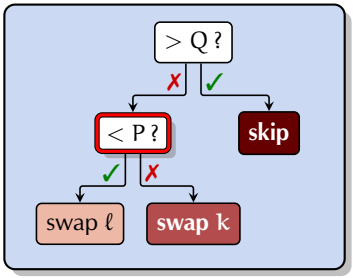


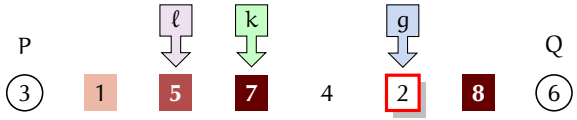
Invariant:



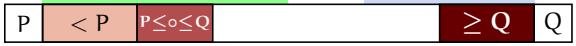
YBB Partitioning Example





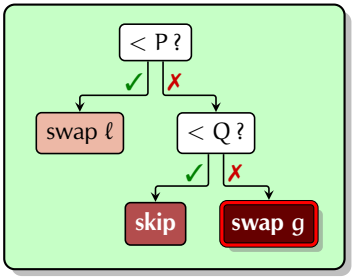


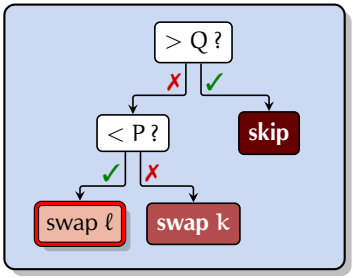
Invariant:



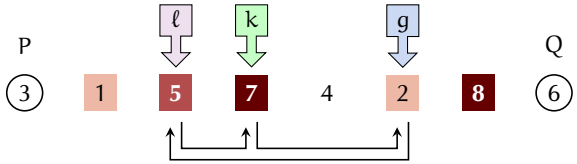

```
YBBQuicksort(A, left, right)
 1  if right ⩽ left then return
 2  P = A[left];   Q = A[right]
 3  if P > Q then Swap P and Q end if
 4  ℓ = left + 1;   g = right − 1;   k = ℓ
 5  while k ⩽ g
 6      if A[k] < P
 7          Swap A[k] and A[ℓ] ;  ℓ = ℓ + 1
 8      else if A[k] ⩾ Q
 9          while A[g] > Q and k < g
10              g = g − 1
11          end while
12          Swap A[k] and A[g] ;  g = g − 1
13          if A[k] < P
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

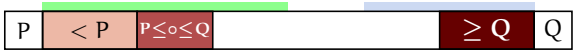




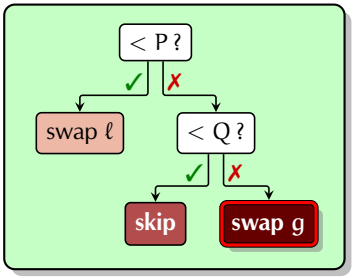

```
YBBQuicksort(A, left, right)
 1  if right ⩽ left then return
 2  P = A[left];   Q = A[right]
 3  if P > Q then Swap P and Q end if
 4  ℓ = left + 1;   g = right − 1;   k = ℓ
 5  while k ⩽ g
 6      if A[k] < P
 7          Swap A[k] and A[ℓ] ;  ℓ = ℓ + 1
 8      else if A[k] ⩾ Q
 9          while A[g] > Q and k < g
10              g = g − 1
11          end while
12          Swap A[k] and A[g] ;  g = g − 1
13          if A[k] < P
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 )
```

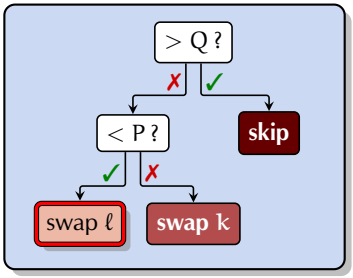


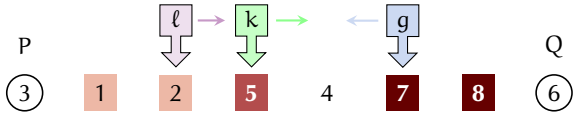
Invariant:



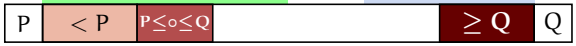
YBB Partitioning Example





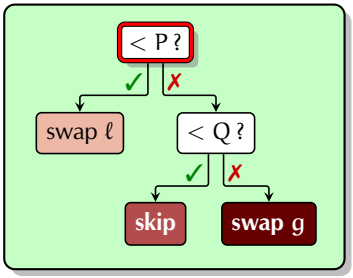


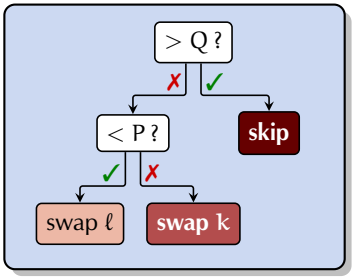
Invariant:

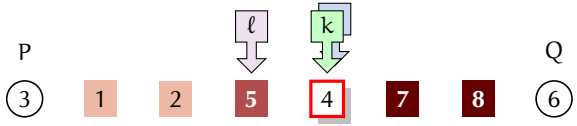



```
YBBQuicksort(A, left, right)
 1  if right ⩽ left then return
 2  P = A[left];   Q = A[right]
 3  if P > Q then Swap P and Q end if
 4  ℓ = left + 1;   g = right − 1;   k = ℓ
 5  while k ⩽ g
 6      if A[k] < P
 7          Swap A[k] and A[ℓ] ; ℓ = ℓ + 1
 8      else if A[k] ⩾ Q
 9          while A[g] > Q and k < g
10              g = g − 1
11          end while
12          Swap A[k] and A[g] ; g = g − 1
13          if A[k] < P
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 )
```

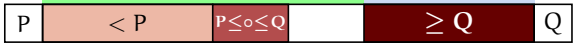
YBB Partitioning Example





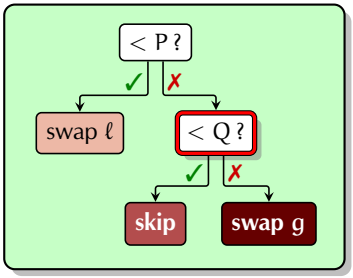


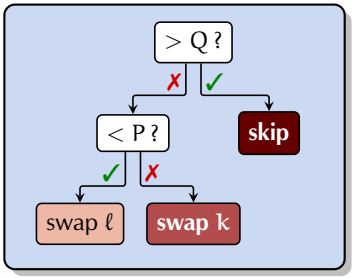
Invariant:

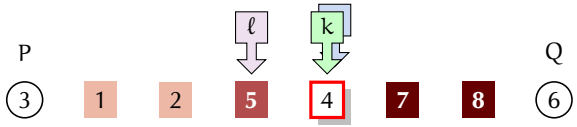



```
YBBQuicksort(A, left, right)
 1  if right ⩽ left then return
 2  P = A[left];   Q = A[right]
 3  if P > Q then Swap P and Q end if
 4  ℓ = left + 1;   g = right − 1;   k = ℓ
 5  while k ⩽ g
 6      if A[k] < P
 7          Swap A[k] and A[ℓ] ; ℓ = ℓ + 1
 8      else if A[k] ⩾ Q
 9          while A[g] > Q and k < g
10              g = g − 1
11          end while
12          Swap A[k] and A[g] ; g = g − 1
13          if A[k] < P
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 )
```

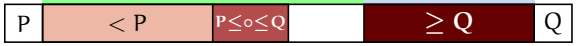
YBB Partitioning Example





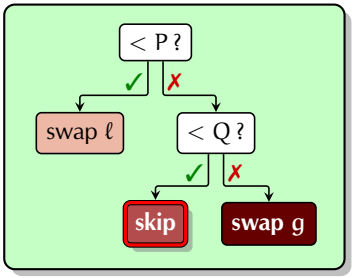


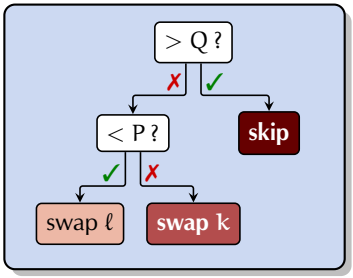
Invariant:

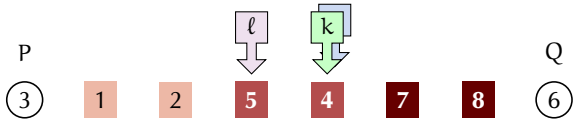



```
YBBQuicksort(A, left, right)
 1  if right ⩽ left then return
 2  P = A[left];   Q = A[right]
 3  if P > Q then Swap P and Q end if
 4  ℓ = left + 1;   g = right − 1;   k = ℓ
 5  while k ⩽ g
 6      if A[k] < P
 7          Swap A[k] and A[ℓ] ; ℓ = ℓ + 1
 8      else if A[k] ⩾ Q
 9          while A[g] > Q and k < g
10              g = g − 1
11          end while
12          Swap A[k] and A[g] ; g = g − 1
13          if A[k] < P
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





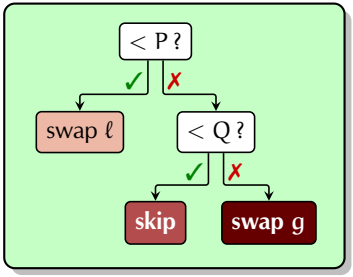


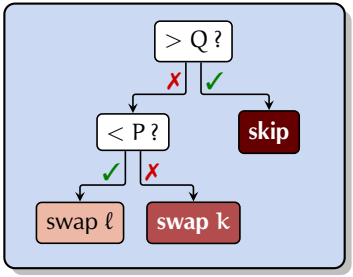
Invariant:




```
YBBQuicksort(A, left, right)
 1  if right ⩽ left then return
 2  P = A[left];   Q = A[right]
 3  if P > Q then Swap P and Q end if
 4  ℓ = left + 1;   g = right − 1;   k = ℓ
 5  while k ⩽ g
 6      if A[k] < P
 7          Swap A[k] and A[ℓ] ; ℓ = ℓ + 1
 8      else if A[k] ⩾ Q
 9          while A[g] > Q and k < g
10              g = g − 1
11          end while
12          Swap A[k] and A[g] ; g = g − 1
13          if A[k] < P
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



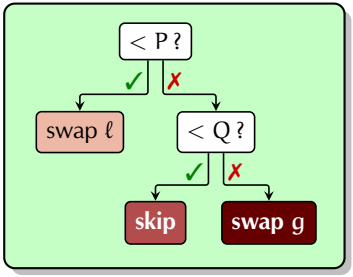


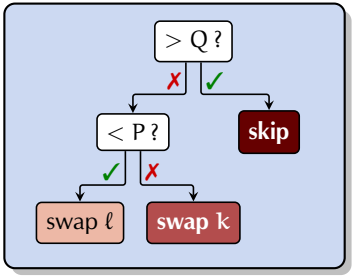


YBBQuicksort(A, *left*, *right*)

```
1  if right ⩽ left then return
2  P = A[left];   Q = A[right]
3  if P > Q then Swap P and Q end if
4  ℓ = left + 1;   g = right − 1;   k = ℓ
5  while k ⩽ g
6      if A[k] < P
7          Swap A[k] and A[ℓ] ; ℓ = ℓ + 1
8      else if A[k] ⩾ Q
9          while A[g] > Q and k < g
10             g = g − 1
11         end while
12         Swap A[k] and A[g] ; g = g − 1
13         if A[k] < P
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

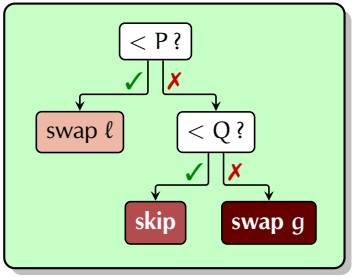


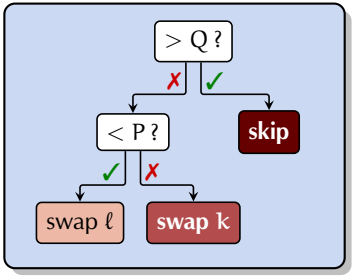





```
YBBQuicksort(A, left, right)
 1  if right ⩽ left then return
 2  P = A[left];   Q = A[right]
 3  if P > Q then Swap P and Q end if
 4  ℓ = left + 1;   g = right − 1;   k = ℓ
 5  while k ⩽ g
 6      if A[k] < P
 7          Swap A[k] and A[ℓ] ; ℓ = ℓ + 1
 8      else if A[k] ⩾ Q
 9          while A[g] > Q and k < g
10              g = g − 1
11          end while
12          Swap A[k] and A[g] ; g = g − 1
13          if A[k] < P
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







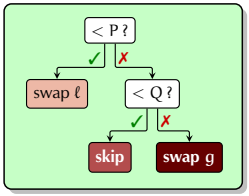
YBBQuicksort(A, *left*, *right*)

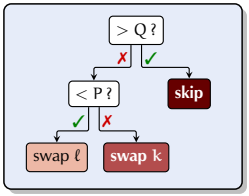
```
1  if right ⩽ left then return
2  P = A[left];   Q = A[right]
3  if P > Q then Swap P and Q end if
4  ℓ = left + 1;   g = right − 1;   k = ℓ
5  while k ⩽ g
6      if A[k] < P
7          Swap A[k] and A[ℓ] ; ℓ = ℓ + 1
8      else if A[k] ⩾ Q
9          while A[g] > Q and k < g
10             g = g − 1
11         end while
12         Swap A[k] and A[g] ; g = g − 1
13         if A[k] < P
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 )
```

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 cmp for small elements,
2 cmps for medium and large.
(Similar if Q used first.)

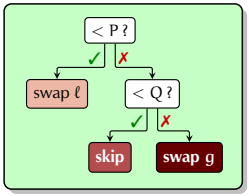
- On average, $\sim \frac{1}{3}n$ elements each kind.

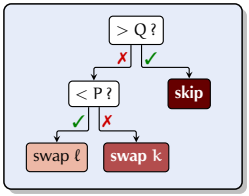
⇝ on average $2 \cdot \frac{2}{3} + 1 \cdot \frac{1}{3} = \frac{5}{3}$ cmps per element

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 cmp for small elements,
2 cmps for medium and large.
(Similar if Q used first.)

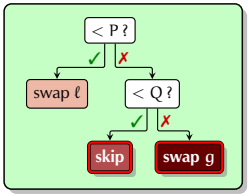
- On average, $\sim \frac{1}{3}n$ elements each kind.

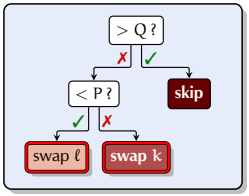
⇝ on average $2 \cdot \frac{2}{3} + 1 \cdot \frac{1}{3} = \frac{5}{3}$ cmps per element

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 cmp for small elements,
2 cmps for medium and large.
(Similar if Q used first.)

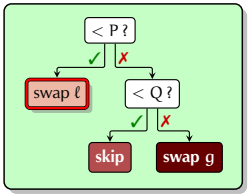
- On average, $\sim \frac{1}{3}n$ elements each kind.

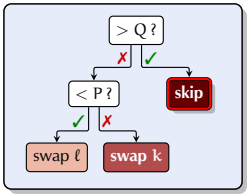
⇝ on average $2 \cdot \frac{2}{3} + 1 \cdot \frac{1}{3} = \frac{5}{3}$ cmps per element

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 cmp for small elements,
2 cmps for medium and large.
(Similar if Q used first.)

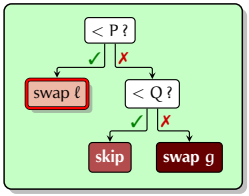
- On average, $\sim \frac{1}{3}n$ elements each kind.

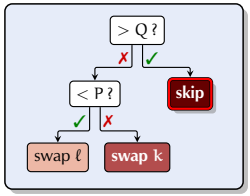
⇝ on average $2 \cdot \frac{2}{3} + 1 \cdot \frac{1}{3} = \frac{5}{3}$ cmps per element

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 cmp for small elements,
2 cmps for medium and large.
(Similar if Q used first.)

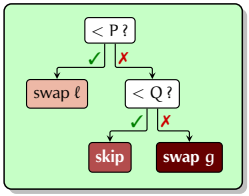
- On average, $\sim \frac{1}{3}n$ elements each kind.

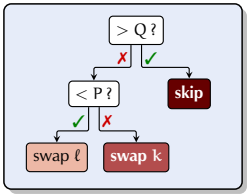
⇝ on average $2 \cdot \frac{2}{3} + 1 \cdot \frac{1}{3} = \frac{5}{3}$ cmps per element

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 cmp for small elements,
2 cmps for medium and large.
(Similar if Q used first.)

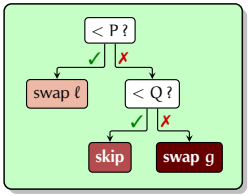
- On average, $\sim \frac{1}{3}n$ elements each kind.

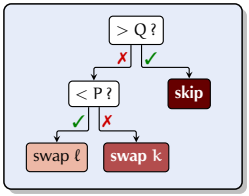
⇝ on average $2 \cdot \frac{2}{3} + 1 \cdot \frac{1}{3} = \frac{5}{3}$ cmps per element

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 cmp for small elements,
2 cmps for medium and large.
(Similar if Q used first.)

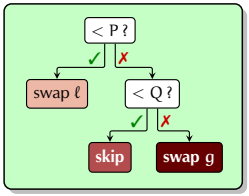
- On average, $\sim \frac{1}{3}n$ elements each kind.

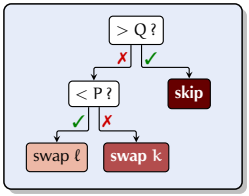
⇝ on average $2 \cdot \frac{2}{3} + 1 \cdot \frac{1}{3} = \frac{5}{3}$ cmps per element

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 cmp for small elements,
2 cmps for medium and large.
(Similar if Q used first.)

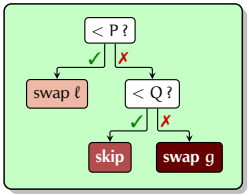
- On average, $\sim \frac{1}{3}n$ elements each kind.

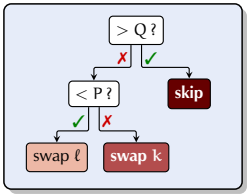
⇝ on average $2 \cdot \frac{2}{3} + 1 \cdot \frac{1}{3} = \frac{5}{3}$ cmps per element

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 cmp for small elements,
2 cmps for medium and large.
(Similar if Q used first.)

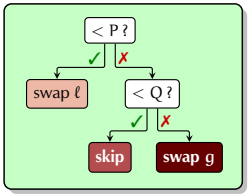
- On average, $\sim \frac{1}{3}n$ elements each kind.

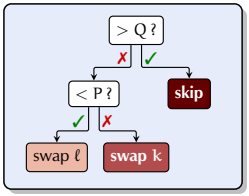
⇝ on average $2 \cdot \frac{2}{3} + 1 \cdot \frac{1}{3} = \frac{5}{3}$ cmps per element

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 cmp for small elements,
2 cmps for medium and large.
(Similar if Q used first.)

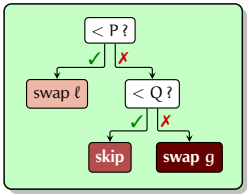
- On average, $\sim \frac{1}{3}n$ elements each kind.

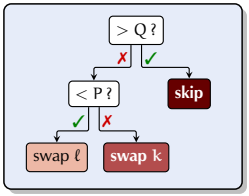
⇝ on average $2 \cdot \frac{2}{3} + 1 \cdot \frac{1}{3} = \frac{5}{3}$ cmps per element

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 cmp for small elements,
2 cmps for medium and large.
(Similar if Q used first.)

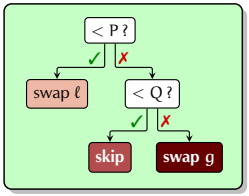
- On average, $\sim \frac{1}{3}n$ elements each kind.

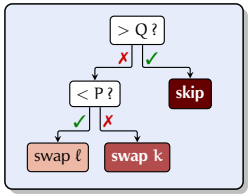
⇝ on average $2 \cdot \frac{2}{3} + 1 \cdot \frac{1}{3} = \frac{5}{3}$ cmps per element

- Sounds plausible?

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 cmp for small elements,
2 cmps for medium and large.
(Similar if Q used first.)

- 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:

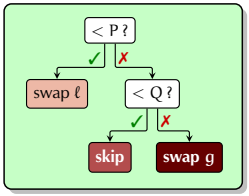
- Have to compare with one pivot first, say P.

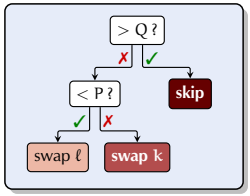
⇝ 1 cmp for small elements,
2 cmps for medium and large.
(Similar if Q used first.)

- 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!





Problem: There are correlations.

- many small elements ⇝ k's range big
- many large elements ⇝ g's range big

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

How many? . . . stay tuned.

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:

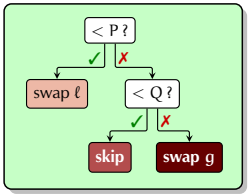
- Have to compare with one pivot first, say P.

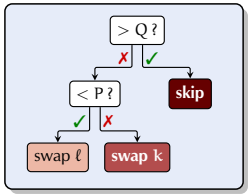
⇝ 1 cmp for small elements,
2 cmps for medium and large.
(Similar if Q used first.)

- 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!





Problem: There are correlations.

- 

many small elements ⇝ k's range big

-

many large elements ⇝ g's range big

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

How many? . . . stay tuned.

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:

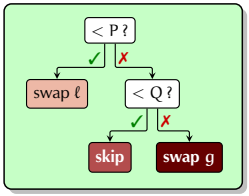
- Have to compare with one pivot first, say P.

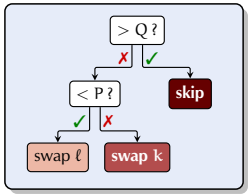
⇝ 1 cmp for small elements,
2 cmps for medium and large.
(Similar if Q used first.)

- 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!

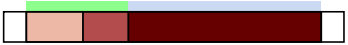




Problem: There are correlations.

- 

many small elements ⇝ k's range big

- 

many large elements ⇝ g's range big

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

How many? . . . stay tuned.

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:

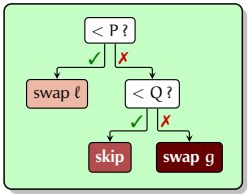
- Have to compare with one pivot first, say P.

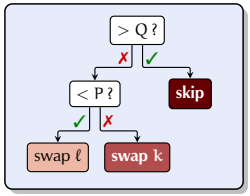
⇝ 1 cmp for small elements,
2 cmps for medium and large.
(Similar if Q used first.)

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

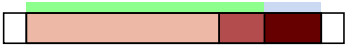
⇝ on average $2 \cdot \frac{2}{3} + 1 \cdot \frac{1}{3} = \frac{5}{3}$ cmps per element

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





Problem: There are correlations.

- 

many small elements ⇝ k's range big

- 

many large elements ⇝ g's range big

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

How many? . . . stay tuned.

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:

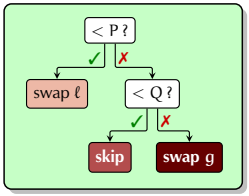
- Have to compare with one pivot first, say P.

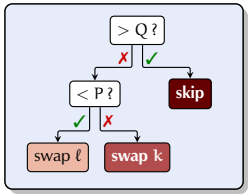
⇝ 1 cmp for small elements,
2 cmps for medium and large.
(Similar if Q used first.)

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

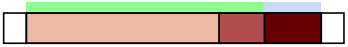
⇝ on average $2 \cdot \frac{2}{3} + 1 \cdot \frac{1}{3} = \frac{5}{3}$ cmps per element

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





Problem: There are correlations.

- 

many small elements ⇝ k's range big

- 

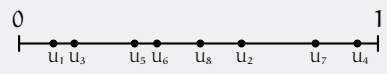
many large elements ⇝ g's range big

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

How many? . . . stay tuned.

Random Model

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



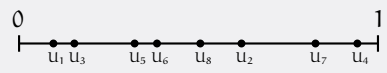
- pairwise **distinct** almost surely
- relative ranking is a **random permutation**
- $\rightsquigarrow$ equivalent to classic model

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

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

Random Model

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



- pairwise **distinct** almost surely
- relative ranking is a **random permutation**

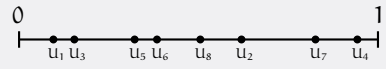
$\rightsquigarrow$ equivalent to classic model

- For **dual-pivot** Quicksort with **fixed** pivots $P \leqslant Q$
  - $\Pr[\ u < P\ ] = D_1$
  - $\Pr[\ P < u < Q\ ] = D_2$
  - $\Pr[\ u > Q\ ] = D_3$

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

Random Model

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



- pairwise **distinct** almost surely
- relative ranking is a **random permutation**

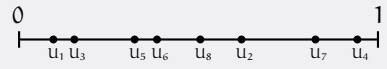
$\rightsquigarrow$ equivalent to classic model

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

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

Random Model

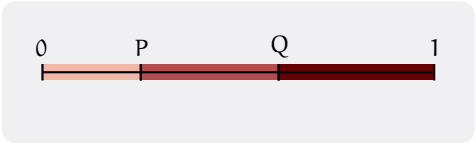
- n **i.i.d.** elements chosen **uniformly** in $(0,1)$



- pairwise **distinct** almost surely
- relative ranking is a **random permutation**

$\rightsquigarrow$ equivalent to classic model

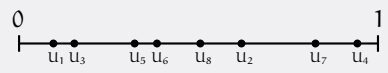
- For **dual-pivot** Quicksort with **fixed** pivots $P \leqslant Q$
  - $\Pr[\, U < P \,] = D_1$
  - $\Pr[\, P < U < Q \,] = D_2$
  - $\Pr[\, U > Q \,] = D_3$



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

Random Model

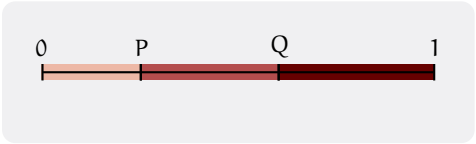
- n **i.i.d.** elements chosen **uniformly** in $(0,1)$



- pairwise **distinct** almost surely
- relative ranking is a **random permutation**

$\rightsquigarrow$ equivalent to classic model

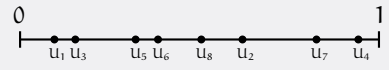
- For **dual-pivot** Quicksort with **fixed** pivots $P \leqslant Q$
  - $\Pr[\, u < P \,] = D_1$
  - $\Pr[\, P < u < Q \,] = D_2$
  - $\Pr[\, u > Q \,] = D_3$



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

Random Model

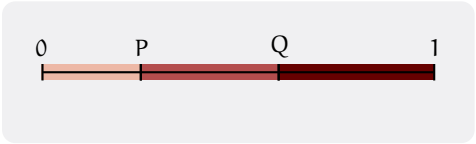
- n **i.i.d.** elements chosen **uniformly** in $(0,1)$



- pairwise **distinct** almost surely
- relative ranking is a **random permutation**

$\rightsquigarrow$ equivalent to classic model

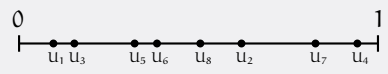
- For **dual-pivot** Quicksort with **fixed** pivots $P \leqslant Q$
  - $\Pr[\, u < P \,] = D_1$
  - $\Pr[\, P < u < Q \,] = D_2$
  - $\Pr[\, u > Q \,] = D_3$



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

Random Model

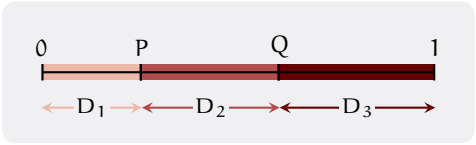
- n **i.i.d.** elements chosen **uniformly** in $(0,1)$



- pairwise **distinct** almost surely
- relative ranking is a **random permutation**

$\rightsquigarrow$ equivalent to classic model

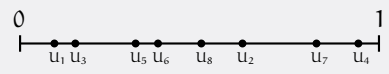
- For **dual-pivot** Quicksort with **fixed** pivots $P \leqslant Q$
  - $\Pr[\, U < P \,] = D_1$
  - $\Pr[\, P < U < Q \,] = D_2$
  - $\Pr[\, U > Q \,] = D_3$



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

Random Model

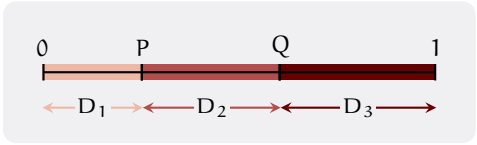
- n **i.i.d.** elements chosen **uniformly** in $(0,1)$



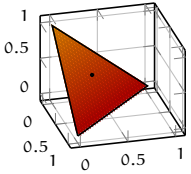
- pairwise **distinct** almost surely
- relative ranking is a **random permutation**

$\rightsquigarrow$ equivalent to classic model

- For **dual-pivot** Quicksort with **fixed** pivots $P \leqslant Q$
  - $\Pr[\, u < P \,] = D_1$
  - $\Pr[\, P < u < Q \,] = D_2$
  - $\Pr[\, u > Q \,] = D_3$

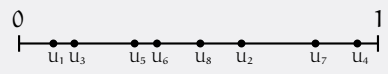


- Known distribution: $\vec{D} = (D_1, D_2, D_3) \overset{\mathcal{D}}{=} \text{Dirichlet}(1,1,1)$.
  (D_1, D_2) uniform in 2D simplex $\{x, y \in (0,1) : x + y < 1\}$



Random Model

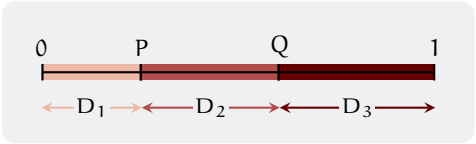
- n **i.i.d.** elements chosen **uniformly** in $(0,1)$



- pairwise **distinct** almost surely
- relative ranking is a **random permutation**

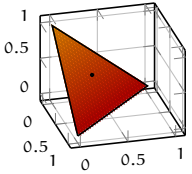
$\rightsquigarrow$ equivalent to classic model

- For **dual-pivot** Quicksort with **fixed** pivots $P \leqslant Q$
  - $\Pr[\, u < P \,] = D_1$
  - $\Pr[\, P < u < Q \,] = D_2$
  - $\Pr[\, u > Q \,] = D_3$



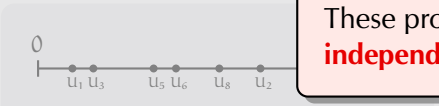
itself a (random) prob. distribution

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



Random Model

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

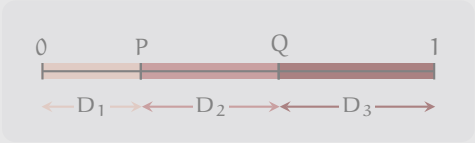


These probabilities hold for all elements U, **independent** of all other elements!

utation

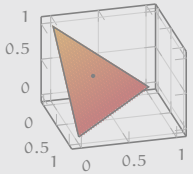
$\rightsquigarrow$ equivalent to classic model

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



itself a (random) prob. distribution

- Known distribution: $\vec{D} = (D_1, D_2, D_3) \overset{\mathcal{D}}{=} \text{Dirichlet}(1,1,1)$.
  (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_1: number of small elements; I_2, I_3 similar
- $\mathcal{K}$: #elements scanned by k; $\mathcal{K} \sim I_1 + I_2$
- $s@\mathcal{K}$: # small elements in k's range; $l@\mathcal{G}$ similar

$\rightsquigarrow\ t(n) \sim 2n - \mathbb{E}[s@\mathcal{K}] - \mathbb{E}[l@\mathcal{G}]$

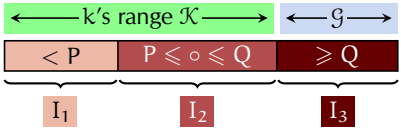
…what is $\mathbb{E}[s@\mathcal{K}]$?

- Consider **fixed** $\vec{D}$,
  $\vec{I} = (I_1, I_2, I_3) \overset{\mathcal{D}}{=} \text{Mult}(n-2, \vec{D})$
$\rightsquigarrow\ \vec{I} = \vec{D}n \pm O(n^{1/2+\varepsilon})$ whp

$\rightsquigarrow\ \mathbb{E}[l@\mathcal{G}] \sim \mathbb{E}\left[D_3 \cdot D_3\right] \cdot n, \qquad \mathbb{E}[s@\mathcal{K}] \sim \mathbb{E}\left[D_1 \cdot (D_1 + D_2)\right] \cdot n$

Comparisons in YBB Partitioning

- $t(n)$: $\mathbb{E}[\#\text{cmps}]$ in **first partitioning step**
- I_1: number of small elements; I_2, I_3 similar
- $\mathcal{K}$: #elements scanned by k; $\mathcal{K} \sim I_1 + I_2$
- $s@\mathcal{K}$: # small elements in k's range; $l@\mathcal{G}$ similar



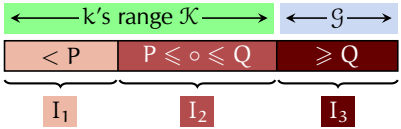
$\rightsquigarrow\ t(n) \sim 2n - \mathbb{E}[s@\mathcal{K}] - \mathbb{E}[l@\mathcal{G}]$ …what is $\mathbb{E}[s@\mathcal{K}]$?

- Consider **fixed** $\vec{D}$,
  $\vec{I} = (I_1, I_2, I_3) \overset{\mathcal{D}}{=} \text{Mult}(n-2, \vec{D})$
- $\rightsquigarrow\ \vec{I} = \vec{D}n \pm O(n^{1/2+\varepsilon})$ whp

$\rightsquigarrow\ \mathbb{E}[l@\mathcal{G}] \sim \mathbb{E}\left[D_3 \cdot D_3\right] \cdot n, \qquad \mathbb{E}[s@\mathcal{K}] \sim \mathbb{E}\left[D_1 \cdot (D_1 + D_2)\right] \cdot n$

Comparisons in YBB Partitioning

- $t(n)$: $\mathbb{E}[\#\text{cmps}]$ in **first partitioning step**
- I_1: number of small elements; I_2, I_3 similar
- $\mathcal{K}$: #elements scanned by k; $\mathcal{K} \sim I_1 + I_2$
- $s@\mathcal{K}$: # small elements in k's range; $l@\mathcal{G}$ similar



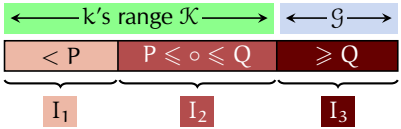
$\rightsquigarrow\ t(n) \sim 2n - \mathbb{E}[s@\mathcal{K}] - \mathbb{E}[l@\mathcal{G}]$ ...what is $\mathbb{E}[s@\mathcal{K}]$?

- Consider **fixed** $\vec{D}$,
  $\vec{I} = (I_1, I_2, I_3) \overset{\mathcal{D}}{=} \text{Mult}(n-2, \vec{D})$
- $\rightsquigarrow\ \vec{I} = \vec{D}n \pm O(n^{1/2+\varepsilon})$ whp

$\rightsquigarrow\ \mathbb{E}[l@\mathcal{G}] \sim \mathbb{E}\left[D_3 \cdot D_3\right] \cdot n, \qquad \mathbb{E}[s@\mathcal{K}] \sim \mathbb{E}\left[D_1 \cdot (D_1 + D_2)\right] \cdot n$

Comparisons in YBB Partitioning

- $t(n)$: $\mathbb{E}[\#\text{cmps}]$ in **first partitioning step**
- I_1: number of small elements; I_2, I_3 similar
- $\mathcal{K}$: #elements scanned by k; $\mathcal{K} \sim I_1 + I_2$
- $s@\mathcal{K}$: # small elements in k's range; $l@\mathcal{G}$ similar



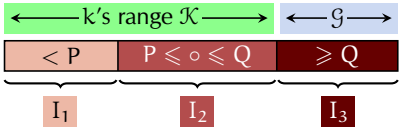
$\rightsquigarrow\ t(n) \sim 2n - \mathbb{E}[s@\mathcal{K}] - \mathbb{E}[l@\mathcal{G}]$ ... what is $\mathbb{E}[s@\mathcal{K}]$?

- Consider **fixed** $\vec{D}$,
  $\vec{I} = (I_1, I_2, I_3) \overset{\mathcal{D}}{=} \text{Mult}(n-2, \vec{D})$
  $\rightsquigarrow\ \vec{I} = \vec{D}n \pm O(n^{1/2+\varepsilon})$ whp

$\rightsquigarrow\ \mathbb{E}[l@\mathcal{G}] \sim \mathbb{E}\left[D_3 \cdot D_3\right] \cdot n$, $\mathbb{E}[s@\mathcal{K}] \sim \mathbb{E}\left[D_1 \cdot (D_1 + D_2)\right] \cdot n$

Comparisons in YBB Partitioning

- $t(n)$: $\mathbb{E}[\#\text{cmps}]$ in **first partitioning step**
- I_1: number of small elements; I_2, I_3 similar
- $\mathcal{K}$: #elements scanned by k; $\mathcal{K} \sim I_1 + I_2$
- $s@\mathcal{K}$: # small elements in k's range; $l@\mathcal{G}$ similar



2 cmps/element, −1 for cheap elements

$$\rightsquigarrow\ t(n) \sim 2n - \mathbb{E}[s@\mathcal{K}] - \mathbb{E}[l@\mathcal{G}]$$

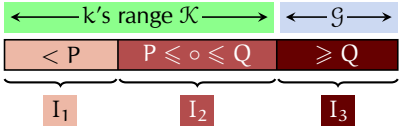
…what is $\mathbb{E}[s@\mathcal{K}]$?

- Consider **fixed** $\vec{D}$,
  $\vec{I} = (I_1, I_2, I_3) \overset{\mathcal{D}}{=} \text{Mult}(n-2, \vec{D})$
  $\rightsquigarrow\ \vec{I} = \vec{D}n \pm O(n^{1/2+\varepsilon})$ whp

$$\rightsquigarrow\ \mathbb{E}[l@\mathcal{G}] \sim \mathbb{E}\left[D_3 \cdot D_3\right] \cdot n, \qquad \mathbb{E}[s@\mathcal{K}] \sim \mathbb{E}\left[D_1 \cdot (D_1 + D_2)\right] \cdot n$$

Comparisons in YBB Partitioning

- $t(n)$: $\mathbb{E}[\#\text{cmps}]$ in **first partitioning step**
- I_1: number of small elements; I_2, I_3 similar
- $\mathcal{K}$: #elements scanned by k; $\mathcal{K} \sim I_1 + I_2$
- $s@\mathcal{K}$: # small elements in k's range; $l@\mathcal{G}$ similar



2 cmps/element, −1 for cheap elements

$$\rightsquigarrow\ t(n) \sim 2n - \mathbb{E}[s@\mathcal{K}] - \mathbb{E}[l@\mathcal{G}]$$

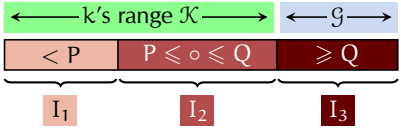
…what is $\mathbb{E}[s@\mathcal{K}]$?

- Consider **fixed** $\vec{D}$,
  $\vec{I} = (I_1, I_2, I_3) \overset{\mathcal{D}}{=} \text{Mult}(n-2, \vec{D})$
  $\rightsquigarrow\ \vec{I} = \vec{D}n \pm O(n^{1/2+\varepsilon})$ whp

$$\rightsquigarrow\ \mathbb{E}[l@\mathcal{G}] \sim \mathbb{E}\left[D_3 \cdot D_3\right] \cdot n, \qquad \mathbb{E}[s@\mathcal{K}] \sim \mathbb{E}\left[D_1 \cdot (D_1 + D_2)\right] \cdot n$$

Comparisons in YBB Partitioning

- $t(n)$: $\mathbb{E}[\#\text{cmps}]$ in **first partitioning step**
- I_1: number of small elements; I_2, I_3 similar
- $\mathcal{K}$: #elements scanned by k; $\mathcal{K} \sim I_1 + I_2$
- $s @ \mathcal{K}$: # small elements in k's range; $l @ \mathcal{G}$ similar

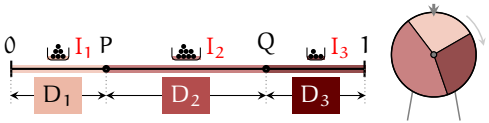


$\rightsquigarrow\ t(n) \sim 2n - \mathbb{E}[s @ \mathcal{K}] - \mathbb{E}[l @ \mathcal{G}]$

…what is $\mathbb{E}[s @ \mathcal{K}]$?

- Consider **fixed** $\vec{D}$, multivariate binomial
  $\vec{I} = (I_1, I_2, I_3) \overset{\mathcal{D}}{=} \text{Mult}(n-2, \vec{D})$

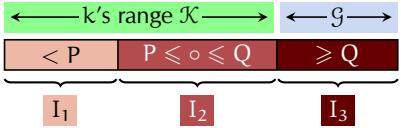
$\rightsquigarrow\ \vec{I} = \vec{D}n \pm O(n^{1/2+\varepsilon})$ whp



$\rightsquigarrow\ \mathbb{E}[l @ \mathcal{G}] \sim \mathbb{E}\left[D_3 \cdot D_3\right] \cdot n, \qquad \mathbb{E}[s @ \mathcal{K}] \sim \mathbb{E}\left[D_1 \cdot (D_1 + D_2)\right] \cdot n$

Comparisons in YBB Partitioning

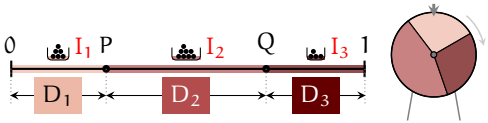
- $t(n)$: $\mathbb{E}[\#\text{cmps}]$ in **first partitioning step**
- I_1: number of small elements; I_2, I_3 similar
- $\mathcal{K}$: #elements scanned by k; $\mathcal{K} \sim I_1 + I_2$
- $s @ \mathcal{K}$: # small elements in k's range; $l @ \mathcal{G}$ similar



$\rightsquigarrow\ t(n) \sim 2n - \mathbb{E}[s @ \mathcal{K}] - \mathbb{E}[l @ \mathcal{G}]$

…what is $\mathbb{E}[s @ \mathcal{K}]$?

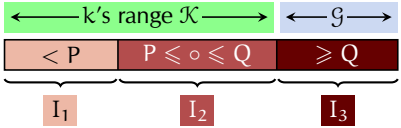
- Consider **fixed** $\vec{D}$, multivariate binomial
  $\vec{I} = (I_1, I_2, I_3) \overset{\mathcal{D}}{=} \text{Mult}(n-2, \vec{D})$
- $\rightsquigarrow\ \vec{I} = \vec{D}n \pm O(n^{1/2+\varepsilon})$ whp ← Chernoff bound



$\rightsquigarrow\ \mathbb{E}[l @ \mathcal{G}] \sim \mathbb{E}\left[D_3 \cdot D_3\right] \cdot n, \qquad \mathbb{E}[s @ \mathcal{K}] \sim \mathbb{E}\left[D_1 \cdot (D_1 + D_2)\right] \cdot n$

Comparisons in YBB Partitioning

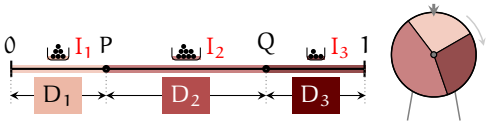
- $t(n)$: $\mathbb{E}[\#\text{cmps}]$ in **first partitioning step**
- I_1: number of small elements; I_2, I_3 similar
- $\mathcal{K}$: #elements scanned by k; $\mathcal{K} \sim I_1 + I_2$
- $s @ \mathcal{K}$: # small elements in k's range; $l @ \mathcal{G}$ similar



$\rightsquigarrow\ t(n) \sim 2n - \mathbb{E}[s @ \mathcal{K}] - \mathbb{E}[l @ \mathcal{G}]$

…what is $\mathbb{E}[s @ \mathcal{K}]$?

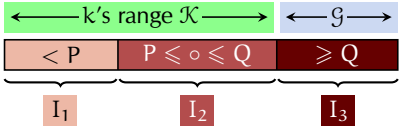
- Consider **fixed** $\vec{D}$, multivariate binomial
  $\vec{I} = (I_1, I_2, I_3) \overset{\mathcal{D}}{=} \text{Mult}(n-2, \vec{D})$
$\rightsquigarrow\ \vec{I} = \vec{D}n \pm O(n^{1/2+\varepsilon})$ whp ← Chernoff bound



$\rightsquigarrow\ \mathbb{E}[l @ \mathcal{G}] \sim \mathbb{E}\left[D_3 \cdot D_3\right] \cdot n, \qquad \mathbb{E}[s @ \mathcal{K}] \sim \mathbb{E}\left[D_1 \cdot (D_1 + D_2)\right] \cdot n$

Comparisons in YBB Partitioning

- $t(n)$: $\mathbb{E}[\#\text{cmps}]$ in **first partitioning step**
- I_1: number of small elements; I_2, I_3 similar
- $\mathcal{K}$: #elements scanned by k; $\mathcal{K} \sim I_1 + I_2$
- $s @ \mathcal{K}$: # small elements in k's range; $\ell @ \mathcal{G}$ similar

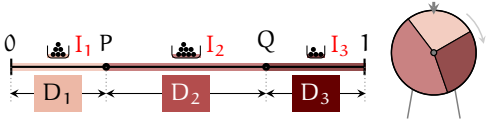


$\rightsquigarrow\ t(n) \sim 2n - \mathbb{E}[s @ \mathcal{K}] - \mathbb{E}[\ell @ \mathcal{G}]$

…what is $\mathbb{E}[s @ \mathcal{K}]$?

- Consider **fixed** $\vec{D}$,
  $\vec{I} = (I_1, I_2, I_3) \overset{\mathcal{D}}{=} \text{Mult}(n-2, \vec{D})$

$\rightsquigarrow\ \vec{I} = \vec{D}n \pm O(n^{1/2+\varepsilon})$ whp

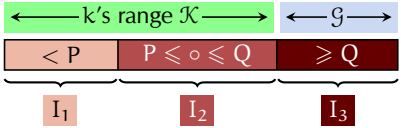


fraction of large elements

$\rightsquigarrow\ \mathbb{E}[\ell @ \mathcal{G}] \sim \mathbb{E}\left[D_3 \cdot D_3\right] \cdot n,$ $\qquad \mathbb{E}[s @ \mathcal{K}] \sim \mathbb{E}\left[D_1 \cdot (D_1 + D_2)\right] \cdot n$

Comparisons in YBB Partitioning

- $t(n)$: $\mathbb{E}[\#\text{cmps}]$ in **first partitioning step**
- I_1: number of small elements; I_2, I_3 similar
- $\mathcal{K}$: #elements scanned by k; $\mathcal{K} \sim I_1 + I_2$
- $s @ \mathcal{K}$: # small elements in k's range; $\ell @ \mathcal{G}$ similar

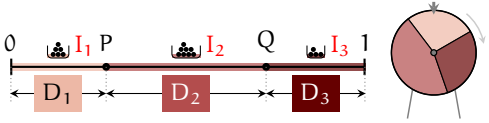


$\rightsquigarrow\ t(n) \sim 2n - \mathbb{E}[s @ \mathcal{K}] - \mathbb{E}[\ell @ \mathcal{G}]$

…what is $\mathbb{E}[s @ \mathcal{K}]$?

- Consider **fixed** $\vec{D}$,
  $\vec{I} = (I_1, I_2, I_3) \overset{\mathcal{D}}{=} \text{Mult}(n-2, \vec{D})$

$\rightsquigarrow\ \vec{I} = \vec{D}n \pm O(n^{1/2+\varepsilon})$ whp



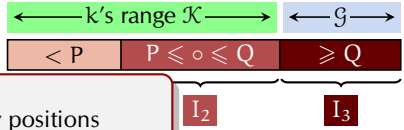
fraction of large elements

fraction of elements in $\mathcal{K}$

$\rightsquigarrow\ \mathbb{E}[\ell @ \mathcal{G}] \sim \mathbb{E}\left[D_3 \cdot D_3\right] \cdot n,$ $\qquad \mathbb{E}[s @ \mathcal{K}] \sim \mathbb{E}\left[D_1 \cdot (D_1 + D_2)\right] \cdot n$

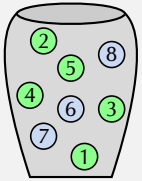
Comparisons in YBB Partitioning

- $t(n)$: $\mathbb{E}[\#\text{cmps}]$ in **first partitioning step**
- I_1: number of small elements; I_2, I_3 similar
- $\mathcal{J}$
- s

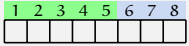


To be precise, we have **conditional** on $\vec{I}$:

$$l@\mathcal{G} \overset{\mathcal{D}}{=} \text{HypG}(n-2;\, I_3,\, I_3)$$



Draw positions of large elements



$\rightsquigarrow$ t …]?

$\rightsquigarrow \mathbb{E}[l@\mathcal{G} \mid \vec{I}] = \dfrac{I_3 \cdot I_3}{n-2}$

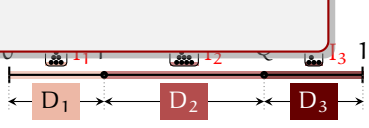
- C

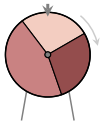
$$\vec{I} = (I_1, I_2, I_3) \overset{\mathcal{D}}{=} \text{Mult}(n-2, \vec{D})$$

$$\rightsquigarrow \vec{I} = \vec{D}n \pm O(n^{1/2+\varepsilon}) \text{ whp}$$

fraction of large elements — fraction of elements in $\mathcal{K}$

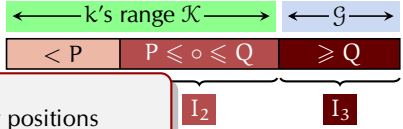
$$\rightsquigarrow \mathbb{E}[l@\mathcal{G}] \sim \mathbb{E}\left[D_3 \cdot D_3\right] \cdot n, \qquad \mathbb{E}[s@\mathcal{K}] \sim \mathbb{E}\left[D_1 \cdot (D_1 + D_2)\right] \cdot n$$





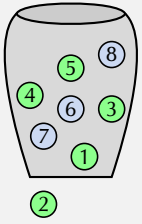
Comparisons in YBB Partitioning

- $t(n)$: $\mathbb{E}[\#\text{cmps}]$ in **first partitioning step**
- I_1: number of small elements; I_2, I_3 similar
- $\mathcal{J}$
- s

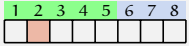


To be precise, we have **conditional** on $\vec{I}$:

$$l@\mathcal{G} \overset{\mathcal{D}}{=} \text{HypG}(n-2; I_3, I_3)$$



Draw positions of large elements



$\rightsquigarrow$ t ...]?

$\rightsquigarrow \mathbb{E}[l@\mathcal{G} \mid \vec{I}] = \frac{I_3 \cdot I_3}{n-2}$

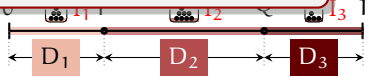
- C

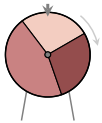
$$\vec{I} = (I_1, I_2, I_3) \overset{\mathcal{D}}{=} \text{Mult}(n-2, \vec{D})$$

$\rightsquigarrow \vec{I} = \vec{D}n \pm O(n^{1/2+\varepsilon})$ whp

fraction of large elements — fraction of elements in $\mathcal{K}$

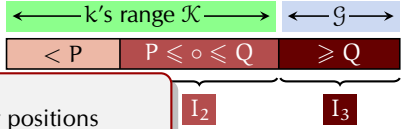
$\rightsquigarrow \mathbb{E}[l@\mathcal{G}] \sim \mathbb{E}\left[D_3 \cdot D_3\right] \cdot n, \qquad \mathbb{E}[s@\mathcal{K}] \sim \mathbb{E}\left[D_1 \cdot (D_1 + D_2)\right] \cdot n$





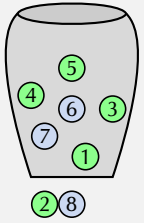
Comparisons in YBB Partitioning

- $t(n)$: $\mathbb{E}[\#\text{cmps}]$ in **first partitioning step**
- I_1: number of small elements; I_2, I_3 similar
- $\mathcal{J}$
- s

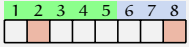


To be precise, we have **conditional** on $\vec{I}$:

$$l@\mathcal{G} \overset{\mathcal{D}}{=} \text{HypG}(n-2;\, I_3,\, I_3)$$



Draw positions of large elements



$\rightsquigarrow$ t …]?

$\rightsquigarrow \mathbb{E}[l@\mathcal{G} \mid \vec{I}] = \dfrac{I_3 \cdot I_3}{n-2}$

- C

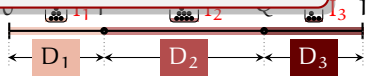
$\vec{I} = (I_1, I_2, I_3) \overset{\mathcal{D}}{=} \text{Mult}(n-2, \vec{D})$

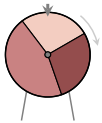
$\rightsquigarrow \vec{I} = \vec{D}n \pm O(n^{1/2+\varepsilon})$ whp

fraction of large elements

fraction of elements in $\mathcal{K}$

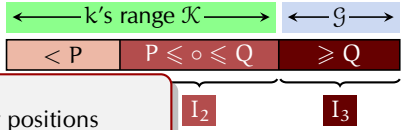
$\rightsquigarrow \mathbb{E}[l@\mathcal{G}] \sim \mathbb{E}\left[D_3 \cdot D_3\right] \cdot n, \qquad \mathbb{E}[s@\mathcal{K}] \sim \mathbb{E}\left[D_1 \cdot (D_1 + D_2)\right] \cdot n$





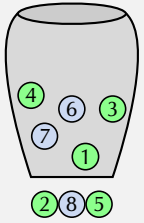
Comparisons in YBB Partitioning

- $t(n)$: $\mathbb{E}[\#\text{cmps}]$ in **first partitioning step**
- I_1: number of small elements; I_2, I_3 similar
- $\mathcal{J}$
- s

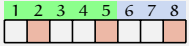


To be precise, we have **conditional** on $\vec{I}$:

$$l@\mathcal{G} \overset{\mathcal{D}}{=} \text{HypG}(n-2;\, I_3,\, I_3)$$



Draw positions of large elements



$\rightsquigarrow$ t …]?

$\rightsquigarrow \mathbb{E}[l@\mathcal{G} \mid \vec{I}] = \dfrac{I_3 \cdot I_3}{n-2}$

- C

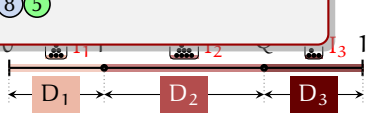
$$\vec{I} = (I_1, I_2, I_3) \overset{\mathcal{D}}{=} \text{Mult}(n-2, \vec{D})$$

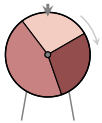
$\rightsquigarrow \vec{I} = \vec{D}n \pm O(n^{1/2+\varepsilon})$ whp

fraction of large elements

fraction of elements in $\mathcal{K}$

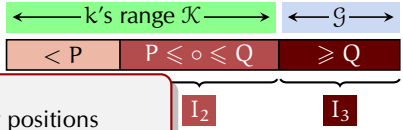
$\rightsquigarrow \mathbb{E}[l@\mathcal{G}] \sim \mathbb{E}\left[D_3 \cdot D_3\right] \cdot n, \qquad \mathbb{E}[s@\mathcal{K}] \sim \mathbb{E}\left[D_1 \cdot (D_1 + D_2)\right] \cdot n$





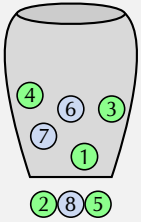
Comparisons in YBB Partitioning

- $t(n)$: $\mathbb{E}[\#\text{cmps}]$ in **first partitioning step**
- I_1: number of small elements; I_2, I_3 similar
- $\mathcal{J}$
- s

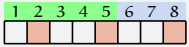


To be precise, we have **conditional** on $\vec{I}$:

$l@\mathcal{G} \overset{\mathcal{D}}{=} \text{HypG}(n-2; I_3, I_3)$



Draw positions of large elements



$l@\mathcal{G} = 1$

$\rightsquigarrow$ t …]?

$\rightsquigarrow \mathbb{E}[l@\mathcal{G} \mid \vec{I}] = \dfrac{I_3 \cdot I_3}{n-2}$

- C

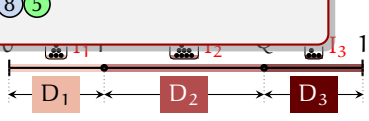
$\vec{I} = (I_1, I_2, I_3) \overset{\mathcal{D}}{=} \text{Mult}(n-2, \vec{D})$

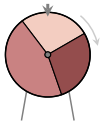
$\rightsquigarrow \vec{I} = \vec{D}n \pm O(n^{1/2+\varepsilon})$ whp

fraction of large elements

fraction of elements in $\mathcal{K}$

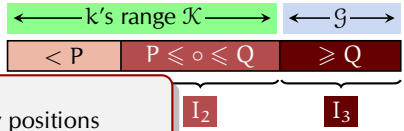
$\rightsquigarrow \mathbb{E}[l@\mathcal{G}] \sim \mathbb{E}\left[D_3 \cdot D_3\right] \cdot n, \qquad \mathbb{E}[s@\mathcal{K}] \sim \mathbb{E}\left[D_1 \cdot (D_1 + D_2)\right] \cdot n$





Comparisons in YBB Partitioning

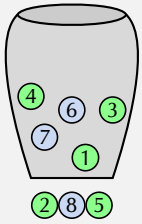
- $t(n)$: $\mathbb{E}[\#\text{cmps}]$ in **first partitioning step**
- I_1: number of small elements; I_2, I_3 similar
- $\mathcal{J}$
- s



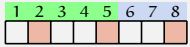
To be precise, we have **conditional** on $\vec{I}$:

$$\mathtt{l}@\mathcal{G} \overset{\mathcal{D}}{=} \mathrm{HypG}(n-2;\, I_3,\, I_3)$$

$$\rightsquigarrow \mathbb{E}[\mathtt{l}@\mathcal{G} \mid \vec{I}] = \frac{I_3 \cdot I_3}{n-2}$$



Draw positions of large elements



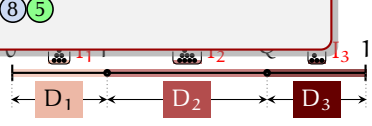
$\mathtt{l}@\mathcal{G} = 1$

$\rightsquigarrow$ t …]?

- C

$$\vec{I} = (I_1, I_2, I_3) \overset{\mathcal{D}}{=} \mathrm{Mult}(n-2, \vec{D})$$

$$\rightsquigarrow \vec{I} = \vec{D}n \pm O(n^{1/2+\varepsilon}) \text{ whp}$$



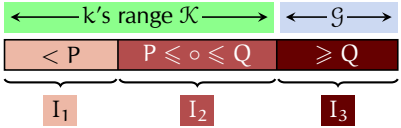
fraction of large elements

fraction of elements in $\mathcal{K}$

$$\rightsquigarrow \mathbb{E}[\mathtt{l}@\mathcal{G}] \sim \mathbb{E}\left[D_3 \cdot D_3\right] \cdot n, \qquad \mathbb{E}[s@\mathcal{K}] \sim \mathbb{E}\left[D_1 \cdot (D_1 + D_2)\right] \cdot n$$

Comparisons in YBB Partitioning

- $t(n)$: $\mathbb{E}[\#\text{cmps}]$ in **first partitioning step**
- I_1: number of small elements; I_2, I_3 similar
- $\mathcal{K}$: #elements scanned by k; $\mathcal{K} \sim I_1 + I_2$
- $s @ \mathcal{K}$: # small elements in k's range; $\ell @ \mathcal{G}$ similar

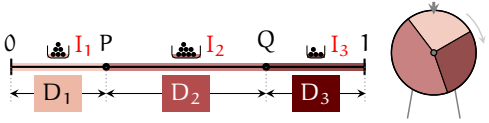


$\rightsquigarrow\ t(n) \sim 2n - \mathbb{E}[s @ \mathcal{K}] - \mathbb{E}[\ell @ \mathcal{G}]$

…what is $\mathbb{E}[s @ \mathcal{K}]$?

- Consider **fixed** $\vec{D}$,
  $\vec{I} = (I_1, I_2, I_3) \overset{\mathcal{D}}{=} \text{Mult}(n-2, \vec{D})$

$\rightsquigarrow\ \vec{I} = \vec{D}n \pm O(n^{1/2+\varepsilon})$ whp



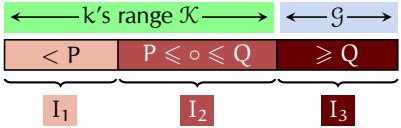
fraction of large elements

fraction of elements in $\mathcal{K}$

$\rightsquigarrow\ \mathbb{E}[\ell @ \mathcal{G}] \sim \mathbb{E}\left[D_3 \cdot D_3\right] \cdot n,$ $\qquad \mathbb{E}[s @ \mathcal{K}] \sim \mathbb{E}\left[D_1 \cdot (D_1 + D_2)\right] \cdot n$

Comparisons in YBB Partitioning

- $t(n)$: $\mathbb{E}[\#\text{cmps}]$ in **first partitioning step**
- I_1: number of small elements; I_2, I_3 similar
- $\mathcal{K}$: #elements scanned by k; $\mathcal{K} \sim I_1 + I_2$
- $s@\mathcal{K}$: # small elements in k's range; $l@\mathcal{G}$ similar

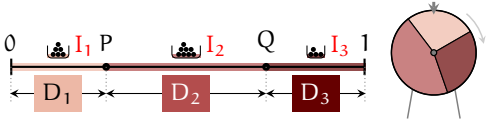


$\rightsquigarrow\ t(n) \sim 2n - \mathbb{E}[s@\mathcal{K}] - \mathbb{E}[l@\mathcal{G}]$

…what is $\mathbb{E}[s@\mathcal{K}]$?

- Consider **fixed** $\vec{D}$,
  $\vec{I} = (I_1, I_2, I_3) \overset{\mathcal{D}}{=} \text{Mult}(n-2, \vec{D})$

$\rightsquigarrow\ \vec{I} = \vec{D}n \pm O(n^{1/2+\varepsilon})$ whp



fraction of large elements

fraction of elements in $\mathcal{K}$

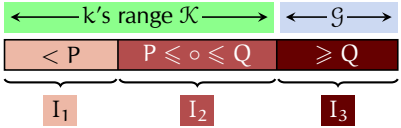
$\rightsquigarrow\ \mathbb{E}[l@\mathcal{G}] \sim \mathbb{E}\left[D_3 \cdot D_3\right] \cdot n,$

$= \frac{1}{6} > \frac{1}{3} \cdot \frac{1}{3}$

$\mathbb{E}[s@\mathcal{K}] \sim \mathbb{E}\left[D_1 \cdot (D_1 + D_2)\right] \cdot n$

Comparisons in YBB Partitioning

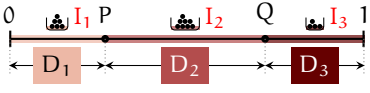
- $t(n)$: $\mathbb{E}[\#\text{cmps}]$ in **first partitioning step**
- I_1: number of small elements; I_2, I_3 similar
- $\mathcal{K}$: #elements scanned by k; $\mathcal{K} \sim I_1 + I_2$
- $s@\mathcal{K}$: # small elements in k's range; $l@\mathcal{G}$ similar

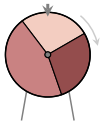


$\rightsquigarrow\ t(n) \sim 2n - \mathbb{E}[s@\mathcal{K}] - \mathbb{E}[l@\mathcal{G}]$ …what is $\mathbb{E}[s@\mathcal{K}]$?

- Consider **fixed** $\vec{D}$,
  $\vec{I} = (I_1, I_2, I_3) \overset{\mathcal{D}}{=} \text{Mult}(n-2, \vec{D})$

$\rightsquigarrow\ \vec{I} = \vec{D}n \pm O(n^{1/2+\varepsilon})$ whp





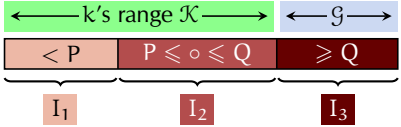
fraction of large elements; fraction of elements in $\mathcal{K}$

$\rightsquigarrow\ \mathbb{E}[l@\mathcal{G}] \sim \mathbb{E}\Big[D_3 \cdot D_3\Big] \cdot n, \quad \mathbb{E}[s@\mathcal{K}] \sim \mathbb{E}\Big[D_1 \cdot (D_1 + D_2)\Big] \cdot n$

$= \frac{1}{6} > \frac{1}{3} \cdot \frac{1}{3}$

Comparisons in YBB Partitioning

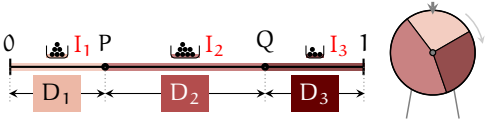
- $t(n)$: $\mathbb{E}[\#\text{cmps}]$ in **first partitioning step**
- I_1: number of small elements; I_2, I_3 similar
- $\mathcal{K}$: #elements scanned by k; $\mathcal{K} \sim I_1 + I_2$
- $s@\mathcal{K}$: # small elements in k's range; $l@\mathcal{G}$ similar



$\rightsquigarrow\ t(n) \sim 2n - \mathbb{E}[s@\mathcal{K}] - \mathbb{E}[l@\mathcal{G}]$ …what is $\mathbb{E}[s@\mathcal{K}]$?

- Consider **fixed** $\vec{D}$,
  $\vec{I} = (I_1, I_2, I_3) \overset{\mathcal{D}}{=} \text{Mult}(n-2, \vec{D})$

$\rightsquigarrow\ \vec{I} = \vec{D}n \pm O(n^{1/2+\varepsilon})$ whp



fraction of large elements

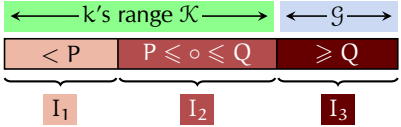
fraction of elements in $\mathcal{K}$

$\rightsquigarrow\ \mathbb{E}[l@\mathcal{G}] \sim \mathbb{E}\left[D_3 \cdot D_3\right] \cdot n, \quad = \frac{1}{6} > \frac{1}{3} \cdot \frac{1}{3}$

$\mathbb{E}[s@\mathcal{K}] \sim \mathbb{E}\left[D_1 \cdot (D_1 + D_2)\right] \cdot n \quad = \frac{1}{4} > \frac{1}{3} \cdot \frac{2}{3}$

Comparisons in YBB Partitioning

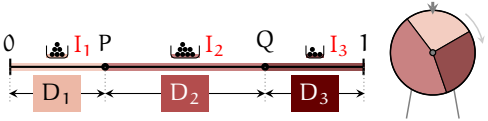
- $t(n)$: $\mathbb{E}[\#\text{cmps}]$ in **first partitioning step**
- I_1: number of small elements; I_2, I_3 similar
- $\mathcal{K}$: #elements scanned by k; $\mathcal{K} \sim I_1 + I_2$
- $s@\mathcal{K}$: # small elements in k's range; $l@\mathcal{G}$ similar



$\rightsquigarrow\ t(n) \sim 2n - \mathbb{E}[s@\mathcal{K}] - \mathbb{E}[l@\mathcal{G}] \sim \frac{19}{12}n$ …what is $\mathbb{E}[s@\mathcal{K}]$?

- Consider **fixed** $\vec{D}$,
  $\vec{I} = (I_1, I_2, I_3) \overset{\mathcal{D}}{=} \text{Mult}(n-2, \vec{D})$

$\rightsquigarrow\ \vec{I} = \vec{D}n \pm O(n^{1/2+\varepsilon})$ whp



fraction of large elements

fraction of elements in $\mathcal{K}$

$\rightsquigarrow\ \mathbb{E}[l@\mathcal{G}] \sim \mathbb{E}\left[D_3 \cdot D_3\right] \cdot n, \quad \mathbb{E}[s@\mathcal{K}] \sim \mathbb{E}\left[D_1 \cdot (D_1 + D_2)\right] \cdot n$

$= \frac{1}{6} > \frac{1}{3} \cdot \frac{1}{3}$ $\qquad$ $= \frac{1}{4} > \frac{1}{3} \cdot \frac{2}{3}$

Recurrence

- $c(n)$: $\mathbb{E}[\#\text{cmps}]$ for sorting

$$\rightsquigarrow \quad c(n) \;=\; t(n) + \mathbb{E}\Big[c(I_1)\Big] + \mathbb{E}\Big[c(I_2)\Big] + \mathbb{E}\Big[c(I_3)\Big]$$

Solve recurrence **without** error term by ***ansatz:***

$$c(n) = \alpha n \ln(n)$$

$$\alpha n \ln(n) \;=\; \tfrac{19}{12} n \;+\; \alpha n \ln(n) \sum_{r=1}^{3} \mathbb{E}[D_r] + \alpha n \sum_{r=1}^{3} \mathbb{E}[D_r \ln(D_r)]$$

- indeed (by induction): $c(n) = \frac{19}{10} n \ln n \pm O(n)$.

Recurrence

- $c(n)$: $\mathbb{E}[\#\text{cmps}]$ for sorting

$$\rightsquigarrow \quad c(n) \;=\; \underbrace{t(n)}_{\sim \frac{19}{12}n} + \mathbb{E}\Big[c(I_1)\Big] + \mathbb{E}\Big[c(I_2)\Big] + \mathbb{E}\Big[c(I_3)\Big]$$

Solve recurrence **without** error term by ***ansatz:***

$$c(n) = \alpha n \ln(n)$$

$$\alpha n \ln(n) \;=\; \tfrac{19}{12}n \;+\; \alpha n \ln(n) \sum_{r=1}^{3} \mathbb{E}[D_r] + \alpha n \sum_{r=1}^{3} \mathbb{E}[D_r \ln(D_r)]$$

- indeed (by induction): $c(n) \;=\; \frac{19}{10} n \ln n \pm O(n)$.

Recurrence

- $c(n)$: $\mathbb{E}[\#\text{cmps}]$ for sorting

$\sim \frac{19}{12}n$ — concentrated around $D_1 n$

$$\rightsquigarrow \quad c(n) \;=\; t(n) + \mathbb{E}\Big[c(I_1)\Big] + \mathbb{E}\Big[c(I_2)\Big] + \mathbb{E}\Big[c(I_3)\Big]$$

Solve recurrence **without** error term by ***ansatz:***

$$c(n) = \alpha n \ln(n)$$

$$\alpha n \ln(n) \;=\; \tfrac{19}{12}n + \alpha n \ln(n) \sum_{r=1}^{3} \mathbb{E}[D_r] + \alpha n \sum_{r=1}^{3} \mathbb{E}[D_r \ln(D_r)]$$

- indeed (by induction): $c(n) = \frac{19}{10} n \ln n \pm O(n)$.

Recurrence

- $c(n)$: $\mathbb{E}[\#\text{cmps}]$ for sorting

$\sim \frac{19}{12}n$ concentrated around $D_1 n$

$$\rightsquigarrow \quad c(n) = t(n) + \mathbb{E}\Big[c(I_1)\Big] + \mathbb{E}\Big[c(I_2)\Big] + \mathbb{E}\Big[c(I_3)\Big]$$

$$= \frac{19}{12}n + \mathbb{E}\Big[c(D_1 \cdot n)\Big] + \mathbb{E}\Big[c(D_2 \cdot n)\Big] + \mathbb{E}\Big[c(D_3 \cdot n)\Big] \pm O(n^{1/2+\varepsilon})$$

Solve recurrence **without** error term by ***ansatz:***

$$c(n) = \alpha n \ln(n)$$

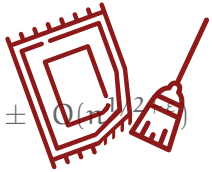
$$\alpha n \ln(n) = \frac{19}{12}n + \alpha n \ln(n) \sum_{r=1}^{3} \mathbb{E}[D_r] + \alpha n \sum_{r=1}^{3} \mathbb{E}[D_r \ln(D_r)]$$

- indeed (by induction): $c(n) = \frac{19}{10} n \ln n \pm O(n)$.

Recurrence

- $c(n)$: $\mathbb{E}[\#\text{cmps}]$ for sorting

$$\rightsquigarrow \quad c(n) = t(n) + \mathbb{E}\Big[c(I_1)\Big] + \mathbb{E}\Big[c(I_2)\Big] + \mathbb{E}\Big[c(I_3)\Big]$$
$$= \tfrac{19}{12}n + \mathbb{E}\Big[c(D_1 \cdot n)\Big] + \mathbb{E}\Big[c(D_2 \cdot n)\Big] + \mathbb{E}\Big[c(D_3 \cdot n)\Big] \pm O(n)$$



Solve recurrence **without** error term by ***ansatz:***

$$c(n) = \alpha n \ln(n)$$

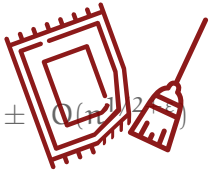
$$\alpha n \ln(n) = \tfrac{19}{12}n + \alpha n \ln(n) \sum_{r=1}^{3} \mathbb{E}[D_r] + \alpha n \sum_{r=1}^{3} \mathbb{E}[D_r \ln(D_r)]$$

- indeed (by induction): $c(n) = \frac{19}{10} n \ln n \pm O(n)$.

Recurrence

- $c(n)$: $\mathbb{E}[\#\text{cmps}]$ for sorting

$$\rightsquigarrow \quad c(n) = t(n) + \mathbb{E}\Big[c(I_1)\Big] + \mathbb{E}\Big[c(I_2)\Big] + \mathbb{E}\Big[c(I_3)\Big]$$
$$= \tfrac{19}{12}n + \mathbb{E}\Big[c(D_1 \cdot n)\Big] + \mathbb{E}\Big[c(D_2 \cdot n)\Big] + \mathbb{E}\Big[c(D_3 \cdot n)\Big] \pm O(n)$$



Solve recurrence **without** error term by ***ansatz***:

$$c(n) = \alpha n \ln(n)$$

($\log_e$)

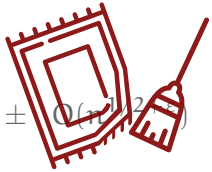
$$\alpha n \ln(n) = \tfrac{19}{12}n + \alpha n \ln(n) \sum_{r=1}^{3} \mathbb{E}[D_r] + \alpha n \sum_{r=1}^{3} \mathbb{E}[D_r \ln(D_r)]$$

- indeed (by induction): $c(n) = \frac{19}{10} n \ln n \pm O(n)$.

Recurrence

- $c(n)$: $\mathbb{E}[\#\text{cmps}]$ for sorting

$$\rightsquigarrow \quad c(n) = t(n) + \mathbb{E}\Big[c(I_1)\Big] + \mathbb{E}\Big[c(I_2)\Big] + \mathbb{E}\Big[c(I_3)\Big]$$
$$= \tfrac{19}{12}n + \mathbb{E}\Big[c(D_1 \cdot n)\Big] + \mathbb{E}\Big[c(D_2 \cdot n)\Big] + \mathbb{E}\Big[c(D_3 \cdot n)\Big] \pm O(n^{1.2+\varepsilon})$$





Solve recurrence **without** error term by ***ansatz***:

$$\boxed{c(n) = \alpha n \ln(n)}$$

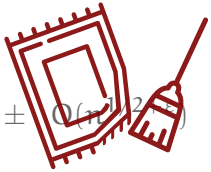
$$\alpha n \ln(n) = \tfrac{19}{12}n + \alpha n \ln(n) \sum_{r=1}^{3} \mathbb{E}[D_r] + \alpha n \sum_{r=1}^{3} \mathbb{E}[D_r \ln(D_r)]$$

- indeed (by induction): $c(n) = \frac{19}{10} n \ln n \pm O(n)$.

Recurrence

- $c(n)$: $\mathbb{E}[\#\text{cmps}]$ for sorting

$$\rightsquigarrow \quad c(n) = t(n) + \mathbb{E}\Big[c(I_1)\Big] + \mathbb{E}\Big[c(I_2)\Big] + \mathbb{E}\Big[c(I_3)\Big]$$
$$= \tfrac{19}{12}n + \mathbb{E}\Big[c(D_1 \cdot n)\Big] + \mathbb{E}\Big[c(D_2 \cdot n)\Big] + \mathbb{E}\Big[c(D_3 \cdot n)\Big] \pm O(n^{1/2+\varepsilon})$$





Solve recurrence **without** error term by ***ansatz***:

$c(n) = \alpha n \ln(n)$

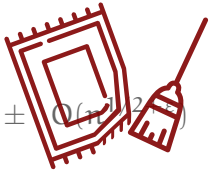
$$\alpha n \ln(n) = \tfrac{19}{12}n + \alpha n \ln(n) \overbrace{\sum_{r=1}^{3} \mathbb{E}[D_r]}^{=1} + \alpha n \sum_{r=1}^{3} \mathbb{E}[D_r \ln(D_r)]$$

- indeed (by induction): $c(n) = \frac{19}{10} n \ln n \pm O(n)$.

Recurrence

- $c(n)$: $\mathbb{E}[\#\text{cmps}]$ for sorting

$$\rightsquigarrow \quad c(n) = t(n) + \mathbb{E}\Big[c(I_1)\Big] + \mathbb{E}\Big[c(I_2)\Big] + \mathbb{E}\Big[c(I_3)\Big]$$
$$= \tfrac{19}{12}n + \mathbb{E}\Big[c(D_1 \cdot n)\Big] + \mathbb{E}\Big[c(D_2 \cdot n)\Big] + \mathbb{E}\Big[c(D_3 \cdot n)\Big] \pm O(n^{1.2\ldots})$$



Solve recurrence **without** error term by ***ansatz***:

$c(n) = \alpha n \ln(n)$

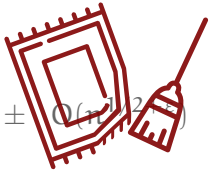
$$\alpha n \ln(n) = \tfrac{19}{12}n + \alpha n \ln(n) \overbrace{\sum_{r=1}^{3} \mathbb{E}[D_r]}^{=1} + \alpha n \sum_{r=1}^{3} \mathbb{E}[D_r \ln(D_r)]$$

- indeed (by induction): $c(n) = \frac{19}{10} n \ln n \pm O(n)$.

Recurrence

- $c(n)$: $\mathbb{E}[\#\text{cmps}]$ for sorting

$$\rightsquigarrow \quad c(n) = t(n) + \mathbb{E}\Big[c(I_1)\Big] + \mathbb{E}\Big[c(I_2)\Big] + \mathbb{E}\Big[c(I_3)\Big]$$
$$= \tfrac{19}{12}n + \mathbb{E}\Big[c(D_1 \cdot n)\Big] + \mathbb{E}\Big[c(D_2 \cdot n)\Big] + \mathbb{E}\Big[c(D_3 \cdot n)\Big] \pm O(n)$$



Solve recurrence **without** error term by ***ansatz***:

$c(n) = \alpha n \ln(n)$

base e Shannon entropy of $\vec{D}$

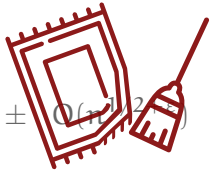
$$\alpha n \ln(n) = \tfrac{19}{12}n + \alpha n \ln(n) \overbrace{\sum_{r=1}^{3} \mathbb{E}[D_r]}^{=1} + \alpha n \overbrace{\sum_{r=1}^{3} \mathbb{E}[D_r \ln(D_r)]}^{=-\mathbb{E}[\mathcal{H}(\vec{D})]}$$

- indeed (by induction): $c(n) = \frac{19}{10} n \ln n \pm O(n)$.

Recurrence

- $c(n)$: $\mathbb{E}[\#\text{cmps}]$ for sorting

$$\rightsquigarrow \quad c(n) = t(n) + \mathbb{E}\Big[c(I_1)\Big] + \mathbb{E}\Big[c(I_2)\Big] + \mathbb{E}\Big[c(I_3)\Big]$$
$$= \tfrac{19}{12}n + \mathbb{E}\Big[c(D_1 \cdot n)\Big] + \mathbb{E}\Big[c(D_2 \cdot n)\Big] + \mathbb{E}\Big[c(D_3 \cdot n)\Big] \pm O(n)$$



Solve recurrence **without** error term by ***ansatz***:

base e Shannon entropy of $\vec{D}$

$c(n) = \alpha n \ln(n)$

$$\alpha n \ln(n) = \tfrac{19}{12}n + \alpha n \ln(n) \overbrace{\sum_{r=1}^{3} \mathbb{E}[D_r]}^{=1} + \alpha n \overbrace{\sum_{r=1}^{3} \mathbb{E}[D_r \ln(D_r)]}^{=-\mathbb{E}[\mathcal{H}(\vec{D})]}$$

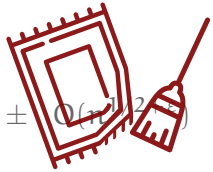
$$\iff \quad \alpha = \frac{\frac{19}{12}}{\mathbb{E}[\mathcal{H}(\vec{D})]}$$

- indeed (by induction): $c(n) = \frac{19}{10} n \ln n \pm O(n)$.

Recurrence

- $c(n)$: $\mathbb{E}[\#\text{cmps}]$ for sorting

$$\rightsquigarrow \quad c(n) = t(n) + \mathbb{E}\Big[c(I_1)\Big] + \mathbb{E}\Big[c(I_2)\Big] + \mathbb{E}\Big[c(I_3)\Big]$$
$$= \tfrac{19}{12}n + \mathbb{E}\Big[c(D_1 \cdot n)\Big] + \mathbb{E}\Big[c(D_2 \cdot n)\Big] + \mathbb{E}\Big[c(D_3 \cdot n)\Big] \pm O(n)$$



Solve recurrence **without** error term by ***ansatz***:

base e Shannon entropy of $\vec{D}$

$$\boxed{c(n) = \alpha n \ln(n)}$$

$$\alpha n \ln(n) = \tfrac{19}{12}n + \alpha n \ln(n) \overbrace{\sum_{r=1}^{3} \mathbb{E}[D_r]}^{=1} + \alpha n \overbrace{\sum_{r=1}^{3} \mathbb{E}[D_r \ln(D_r)]}^{=-\mathbb{E}[\mathcal{H}(\vec{D})]}$$

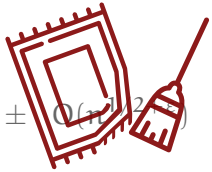
$$\iff \quad \alpha = \frac{\frac{19}{12}}{\mathbb{E}[\mathcal{H}(\vec{D})]} \leftarrow \tfrac{5}{6}$$

- indeed (by induction): $c(n) = \tfrac{19}{10} n \ln n \pm O(n)$.

Recurrence

- $c(n)$: $\mathbb{E}[\#\text{cmps}]$ for sorting

$$\rightsquigarrow \quad c(n) = t(n) + \mathbb{E}\Big[c(I_1)\Big] + \mathbb{E}\Big[c(I_2)\Big] + \mathbb{E}\Big[c(I_3)\Big]$$
$$= \tfrac{19}{12}n + \mathbb{E}\Big[c(D_1 \cdot n)\Big] + \mathbb{E}\Big[c(D_2 \cdot n)\Big] + \mathbb{E}\Big[c(D_3 \cdot n)\Big] \pm O(n)$$



Solve recurrence **without** error term by ***ansatz***:

base e Shannon entropy of $\vec{D}$

$$\boxed{c(n) = \alpha n \ln(n)}$$

$$\alpha n \ln(n) = \tfrac{19}{12}n + \alpha n \ln(n) \overbrace{\sum_{r=1}^{3} \mathbb{E}[D_r]}^{=1} + \alpha n \overbrace{\sum_{r=1}^{3} \mathbb{E}[D_r \ln(D_r)]}^{=-\mathbb{E}[\mathcal{H}(\vec{D})]}$$

$$\iff \quad \alpha = \frac{\frac{19}{12}}{\mathbb{E}[\mathcal{H}(\vec{D})]} \overset{\leftarrow \frac{5}{6}}{=} \frac{19}{10}$$

- indeed (by induction): $c(n) = \frac{19}{10} n \ln n \pm O(n)$.

Recurrence

- $c(n)$: $\mathbb{E}[\#\text{cmps}]$ for sorting

$$\rightsquigarrow \quad c(n) = t(n) + \mathbb{E}\Big[c(I_1)\Big] + \mathbb{E}\Big[c(I_2)\Big] + \mathbb{E}\Big[c(I_3)\Big]$$
$$= \tfrac{19}{12}n + \mathbb{E}\Big[c(D_1 \cdot n)\Big] + \mathbb{E}\Big[c(D_2 \cdot n)\Big] + \mathbb{E}\Big[c(D_3 \cdot n)\Big] \pm O(n^{1/2+\varepsilon})$$



Solve recurrence **without** error term by ***ansatz***:

base e Shannon entropy of $\vec{D}$

$c(n) = \alpha n \ln(n)$

$$\alpha n \ln(n) = \tfrac{19}{12}n + \alpha n \ln(n) \overbrace{\sum_{r=1}^{3} \mathbb{E}[D_r]}^{=1} + \alpha n \overbrace{\sum_{r=1}^{3} \mathbb{E}[D_r \ln(D_r)]}^{=-\mathbb{E}[\mathcal{H}(\vec{D})]}$$
$$\iff \alpha = \frac{\frac{19}{12}}{\mathbb{E}[\mathcal{H}(\vec{D})]} = \tfrac{19}{10} \quad \left(\mathbb{E}[\mathcal{H}(\vec{D})] \leftarrow \tfrac{5}{6}\right)$$

- indeed (by induction): $c(n) = \frac{19}{10} n \ln n \pm O(n)$.

Recurrence

- $c(n)$: $\mathbb{E}[\#\text{cmps}]$ for sorting

$$\rightsquigarrow \quad c(n) = t(n) + \mathbb{E}\Big[c(I_1)\Big] + \mathbb{E}\Big[c(I_2)\Big] + \mathbb{E}\Big[c(I_3)\Big]$$
$$= \tfrac{19}{12}n + \mathbb{E}\Big[c(D_1 \cdot n)\Big] + \mathbb{E}\Big[c(D_2 \cdot n)\Big] + \mathbb{E}\Big[c(D_3 \cdot n)\Big] \pm O(n^{1/2+\varepsilon})$$



Solve recurrence **without** error term by ***ansatz***:

base e Shannon entropy of $\vec{D}$

$$c(n) = \alpha n \ln(n)$$

$$\alpha n \ln(n) = \tfrac{19}{12}n + \alpha n \ln(n) \overbrace{\sum_{r=1}^{3} \mathbb{E}[D_r]}^{=1} + \alpha n \overbrace{\sum_{r=1}^{3} \mathbb{E}[D_r \ln(D_r)]}^{=-\mathbb{E}[\mathcal{H}(\vec{D})]}$$

$$\iff \quad \alpha = \frac{\frac{19}{12}}{\mathbb{E}[\mathcal{H}(\vec{D})]} = \frac{19}{10}$$

$\mathbb{E}[\mathcal{H}(\vec{D})] \leftarrow \frac{5}{6}$

$\rightsquigarrow$ Salvador Roura's *Continuous Master Theorem*

- indeed (by induction): $c(n) = \frac{19}{10} n \ln n \pm O(n)$.

Recurrence

Why does Shannon entropy show up here?

- Must learn $\mathrm{ld}(n!) \sim n \,\mathrm{ld}\, n$ **bits of information** to sort.
- For fixed $\vec{D}$, one classification (small / medium / large)
  - yields $\mathcal{H}_{\mathrm{ld}}(\vec{D}) = -\sum_{r=1}^{3} D_r \,\mathrm{ld}(D_r)$ bits of information
  - costs $a = 2 - \mathbb{E}[D_1 + D_3] - \mathbb{E}[D_1 \,(D_1 + D_2)] = \frac{19}{12}$ cmps

⇝ Average ***information yield***: $\mathcal{I} = \mathbb{E}[\mathcal{H}_{\mathrm{ld}}(\vec{D})] \,/\, a$ bits per comparison

⇝ Need $\dfrac{\mathrm{ld}(n!)}{\mathcal{I}} \sim \dfrac{a}{\mathbb{E}[\mathcal{H}_{\mathrm{ld}}(\vec{D})]} n \,\mathrm{ld}\, n = \dfrac{a}{\mathbb{E}[\mathcal{H}(\vec{D})]} n \ln n$ cmps!

$$\iff \quad \alpha = \frac{\frac{19}{12}}{\mathbb{E}[\mathcal{H}(\vec{D})]} = \frac{19}{10}$$

$\mathbb{E}[\mathcal{H}(\vec{D})] = \frac{5}{6}$

⇝ Salvador Roura's *Continuous Master Theorem*

- indeed (by induction): $c(n) = \frac{19}{10} n \ln n \pm O(n)$.

Recurrence

Why does Shannon entropy show up here?

- Must learn $\log_2 \mathrm{ld}(n!) \sim n \,\mathrm{ld}\, n$ **bits of information** to sort.
- For fixed $\vec{D}$, one classification (small / medium / large)
  - yields $\mathcal{H}_{\mathrm{ld}}(\vec{D}) = -\sum_{r=1}^{3} D_r \,\mathrm{ld}(D_r)$ bits of information
  - costs $a = 2 - \mathbb{E}[D_1 \mid D_3] - \mathbb{E}[D_1 \mid (D_1 + D_2)] = \frac{19}{12}$ cmps

$\rightsquigarrow$ Average ***information yield***: $\mathcal{I} = \mathbb{E}[\mathcal{H}_{\mathrm{ld}}(\vec{D})] \,/\, a$ bits per comparison

$\rightsquigarrow$ Need $\dfrac{\mathrm{ld}(n!)}{\mathcal{I}} \sim \dfrac{a}{\mathbb{E}[\mathcal{H}_{\mathrm{ld}}(\vec{D})]} n \,\mathrm{ld}\, n = \dfrac{a}{\mathbb{E}[\mathcal{H}(\vec{D})]} n \ln n$ cmps!

$\iff$ $\alpha = \dfrac{\frac{19}{12}}{\mathbb{E}[\mathcal{H}(\vec{D})]} = \dfrac{19}{10}$ ($\mathbb{E}[\mathcal{H}(\vec{D})] = \frac{5}{6}$)

$\rightsquigarrow$ Salvador Roura's *Continuous Master Theorem*

- indeed (by induction): $c(n) = \frac{19}{10} n \ln n \pm O(n)$.

Recurrence

Why does Shannon entropy show up here?

- Must learn $\mathrm{ld}(n!) \sim n \,\mathrm{ld}\, n$ **bits of information** to sort.
- For fixed $\vec{D}$, one classification (small / medium / large)
  - yields $\mathcal{H}_{\mathrm{ld}}(\vec{D}) = -\sum_{r=1}^{3} D_r \,\mathrm{ld}(D_r)$ bits of information
  - costs $a = 2 - \mathbb{E}[D_3 \mid D_3] - \mathbb{E}[D_1 \mid (D_1 + D_2)] = \frac{19}{12}$ cmps

⇝ Average ***information yield***: $\mathcal{I} = \mathbb{E}[\mathcal{H}_{\mathrm{ld}}(\vec{D})] \,/\, a$ bits per comparison

⇝ Need $\dfrac{\mathrm{ld}(n!)}{\mathcal{I}} \sim \dfrac{a}{\mathbb{E}[\mathcal{H}_{\mathrm{ld}}(\vec{D})]} n \,\mathrm{ld}\, n = \dfrac{a}{\mathbb{E}[\mathcal{H}(\vec{D})]} n \ln n$ cmps!

$r=1$ $r=1$

$$\iff \quad \alpha = \frac{\frac{19}{12}}{\mathbb{E}[\mathcal{H}(\vec{D})]} = \frac{19}{10} \qquad \left(\mathbb{E}[\mathcal{H}(\vec{D})] = \frac{5}{6}\right)$$

⇝ Salvador Roura's *Continuous Master Theorem*

- indeed (by induction): $c(n) = \frac{19}{10} n \ln n \pm O(n)$.

Why does Shannon entropy show up here?

- Must learn $\mathrm{ld}(n!) \sim n \,\mathrm{ld}\, n$ **bits of information** to sort.
- For fixed $\vec{D}$, one classification (small / medium / large)

binary entropy

- yields $\mathcal{H}_{\mathrm{ld}}(\vec{D}) = -\sum_{r=1}^{3} D_r \,\mathrm{ld}(D_r)$ bits of information
- costs $a = 2 - \mathbb{E}[D_3 \mid D_3] - \mathbb{E}[D_1 \mid (D_1 + D_2)] = \frac{19}{12}$ cmps

⇝ Average ***information yield***: $\mathcal{I} = \mathbb{E}[\mathcal{H}_{\mathrm{ld}}(\vec{D})] \,/\, a$ bits per comparison

⇝ Need $\dfrac{\mathrm{ld}(n!)}{\mathcal{I}} \sim \dfrac{a}{\mathbb{E}[\mathcal{H}_{\mathrm{ld}}(\vec{D})]} n \,\mathrm{ld}\, n = \dfrac{a}{\mathbb{E}[\mathcal{H}(\vec{D})]} n \ln n$ cmps!

$r=1$ $r=1$

$$\iff \quad \alpha = \frac{\frac{19}{12}}{\mathbb{E}[\mathcal{H}(\vec{D})]} = \frac{19}{10}$$

$\mathbb{E}[\mathcal{H}(\vec{D})] = \frac{5}{6}$

⇝ Salvador Roura's *Continuous Master Theorem*

- indeed (by induction): $c(n) = \frac{19}{10} n \ln n \pm O(n)$.

Why does Shannon entropy show up here?

- Must learn $\mathrm{ld}(n!) \sim n \,\mathrm{ld}\, n$ **bits of information** to sort.
- For fixed $\vec{D}$, one classification (small / medium / large)

binary entropy

- yields $\mathcal{H}_{\mathrm{ld}}(\vec{D}) = -\sum_{r=1}^{3} D_r \,\mathrm{ld}(D_r)$ bits of information

definition of entropy!

- costs $a = 2 - \mathbb{E}[D_3 \mid D_3] - \mathbb{E}[D_1 \mid (D_1 + D_2)] = \frac{19}{12}$ cmps

⇝ Average ***information yield***: $\mathcal{I} = \mathbb{E}[\mathcal{H}_{\mathrm{ld}}(\vec{D})] \,/\, a$ bits per comparison

⇝ Need $\dfrac{\mathrm{ld}(n!)}{\mathcal{I}} \sim \dfrac{a}{\mathbb{E}[\mathcal{H}_{\mathrm{ld}}(\vec{D})]} n \,\mathrm{ld}\, n = \dfrac{a}{\mathbb{E}[\mathcal{H}(\vec{D})]} n \ln n$ cmps!

$r=1$ $r=1$

$$\iff \quad \alpha = \frac{\frac{19}{12}}{\mathbb{E}[\mathcal{H}(\vec{D})]} = \frac{19}{10}$$

$\frac{5}{6}$

⇝ Salvador Roura's *Continuous Master Theorem*

- indeed (by induction): $c(n) = \frac{19}{10} n \ln n \pm O(n)$.

Why does Shannon entropy show up here?

- Must learn $\mathrm{ld}(n!) \sim n \,\mathrm{ld}\, n$ **bits of information** to sort.
- For fixed $\vec{D}$, one classification (small / medium / large)
  - yields $\mathcal{H}_{\mathrm{ld}}(\vec{D}) = -\sum_{r=1}^{3} D_r \,\mathrm{ld}(D_r)$ bits of information (binary entropy; definition of entropy!)
  - costs $a = 2 - \mathbb{E}[D_3 \;\; D_3] - \mathbb{E}[D_1 \;\; (D_1 + D_2)] = \frac{19}{12}$ cmps

$\rightsquigarrow$ Average ***information yield***: $\mathcal{I} = \mathbb{E}[\mathcal{H}_{\mathrm{ld}}(\vec{D})] \,/\, a$ bits per comparison

$\rightsquigarrow$ Need $\dfrac{\mathrm{ld}(n!)}{\mathcal{I}} \sim \dfrac{a}{\mathbb{E}[\mathcal{H}_{\mathrm{ld}}(\vec{D})]} n \,\mathrm{ld}\, n = \dfrac{a}{\mathbb{E}[\mathcal{H}(\vec{D})]} n \ln n$ cmps!

$\rightsquigarrow$ Salvador Roura's *Continuous Master Theorem*

$$\iff \quad \alpha = \frac{\frac{19}{12}}{\mathbb{E}[\mathcal{H}(\vec{D})]} = \frac{19}{10} \qquad \left(\mathbb{E}[\mathcal{H}(\vec{D})] = \tfrac{5}{6}\right)$$

- indeed (by induction): $c(n) = \frac{19}{10} n \ln n \pm O(n)$.

Why does Shannon entropy show up here?

- Must learn $\mathrm{ld}(n!) \sim n \,\mathrm{ld}\, n$ **bits of information** to sort.
- For fixed $\vec{D}$, one classification (small / medium / large)
  - yields $\mathcal{H}_{\mathrm{ld}}(\vec{D}) = -\sum_{r=1}^{3} D_r \,\mathrm{ld}(D_r)$ bits of information (binary entropy; definition of entropy!)
  - costs $a = 2 - \mathbb{E}[D_3 \mid D_3] - \mathbb{E}[D_1 \mid (D_1 + D_2)] = \frac{19}{12}$ cmps
- ⇝ Average ***information yield***: $\mathcal{I} = \mathbb{E}[\mathcal{H}_{\mathrm{ld}}(\vec{D})] \,/\, a$ bits per comparison
- ⇝ Need $\frac{\mathrm{ld}(n!)}{\mathcal{I}} \sim \frac{a}{\mathbb{E}[\mathcal{H}_{\mathrm{ld}}(\vec{D})]} n \,\mathrm{ld}\, n = \frac{a}{\mathbb{E}[\mathcal{H}(\vec{D})]} n \ln n$ cmps!

$$\iff \quad \alpha = \frac{\frac{19}{12}}{\mathbb{E}[\mathcal{H}(\vec{D})]} = \frac{19}{10} \qquad \left(\mathbb{E}[\mathcal{H}(\vec{D})] = \frac{5}{6}\right)$$

⇝ Salvador Roura's *Continuous Master Theorem*

- indeed (by induction): $c(n) = \frac{19}{10} n \ln n \pm O(n)$.

Why does Shannon entropy show up here?

- Must learn $\mathrm{ld}(n!) \sim n \,\mathrm{ld}\, n$ **bits of information** to sort.
- For fixed $\vec{D}$, one classification (small / medium / large)
  - yields $\mathcal{H}_{\mathrm{ld}}(\vec{D}) = -\sum_{r=1}^{3} D_r \,\mathrm{ld}(D_r)$ bits of information (binary entropy; definition of entropy!)
  - costs $a = 2 - \mathbb{E}[D_3 \mid D_3] - \mathbb{E}[D_1 \mid (D_1 + D_2)] = \frac{19}{12}$ cmps

⇝ Average ***information yield***: $\mathcal{I} = \mathbb{E}[\mathcal{H}_{\mathrm{ld}}(\vec{D})] \,/\, a$ bits per comparison

⇝ Need $\dfrac{\mathrm{ld}(n!)}{\mathcal{I}} \sim \dfrac{a}{\mathbb{E}[\mathcal{H}_{\mathrm{ld}}(\vec{D})]} n \,\mathrm{ld}\, n = \dfrac{a}{\mathbb{E}[\mathcal{H}(\vec{D})]} n \ln n$ cmps!

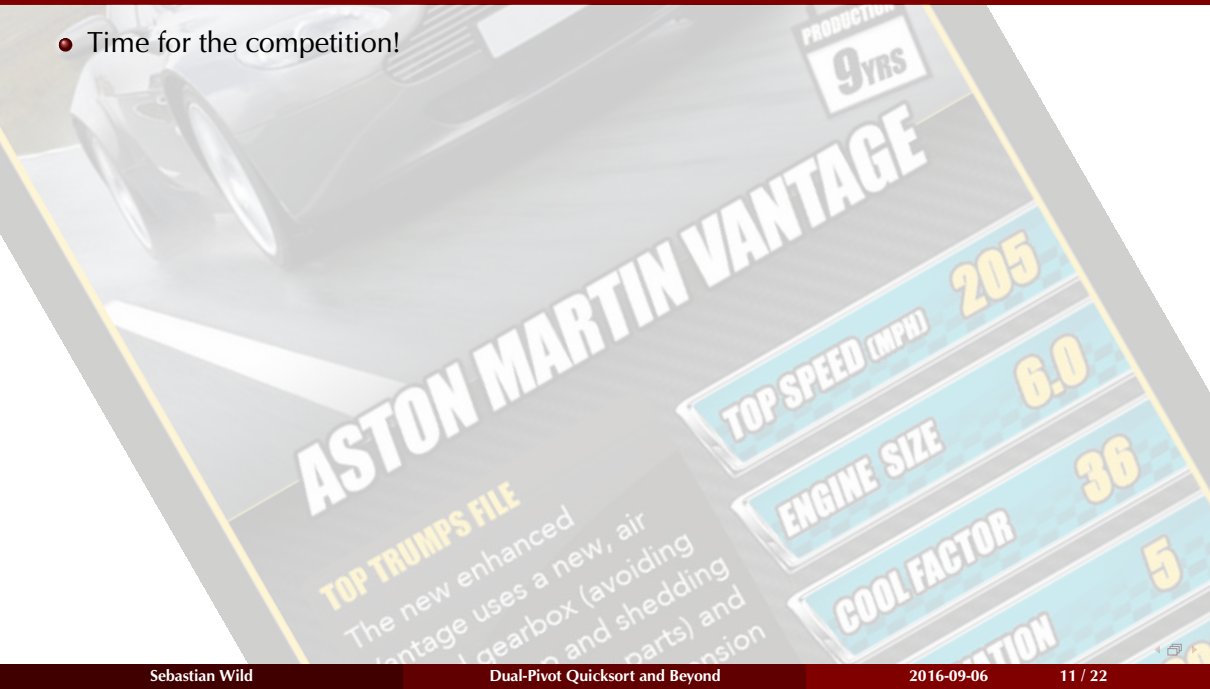
$$\iff \quad \alpha = \frac{\frac{19}{12}}{\mathbb{E}[\mathcal{H}(\vec{D})]} = \frac{19}{10} \qquad \left(\mathbb{E}[\mathcal{H}(\vec{D})] = \frac{5}{6}\right)$$

⇝ Salvador Roura's *Continuous Master Theorem*

- indeed (by induction): $c(n) = \frac{19}{10} n \ln n \pm O(n)$.

Classic vs. YBB Quicksort

- Time for the competition!



Classic vs. YBB Quicksort

- Time for the competition!

	CQS	YQS	Relative
Running Time	(from various experiments)		−10±2%

Classic vs. YBB Quicksort

- Time for the competition!

	CQS	YQS	Relative
Running Time	(from various experiments)		$-10 \pm 2\%$
Comparisons		1.9	

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

Classic vs. YBB Quicksort

- Time for the competition!

	CQS	YQS	Relative
Running Time	(from various experiments)		$-10\pm2\%$
Comparisons	2	1.9	-5%

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

Classic vs. YBB Quicksort

- Time for the competition!

	CQS	YQS	Relative
Running Time	(from various experiments)		−10±2%
Comparisons	2	1.9	−5%

Results for random pivots . . .

Classic vs. YBB Quicksort

- Time for the competition!

	CQS	YQS	Relative
Running Time	(from various experiments)		−10±2%
Comparisons	2	1.9	−5%

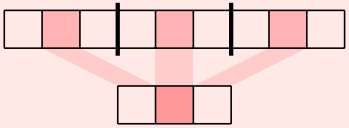
Results for random pivots ... but Java 6/7 use **pivot sampling**:

Classic vs. YBB Quicksort

- Time for the competition!

	CQS	YQS	Relative
Running Time	(from various experiments)		$-10 \pm 2\%$
Comparisons	2	1.9	-5%

Results for random pivots ... but Java 6/7 use **pivot sampling**:



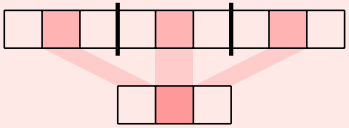
Java 6:
Ninther ← pseudomedian of 9

Classic vs. YBB Quicksort

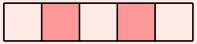
- Time for the competition!

	CQS	YQS	Relative
Running Time	(from various experiments)		−10±2%
Comparisons	2	1.9	−5%

Results for random pivots ... but Java 6/7 use **pivot sampling**:



Java 6:
Ninther ← pseudomedian of 9



Java 7:
Tertiles of 5

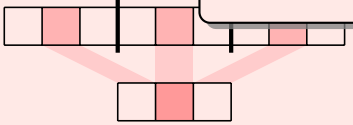
Classic vs. YBB Quicksort

- Time for the competition!

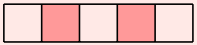
	CQS	YQS	Relative
Running Time	(from various experiments)		−10±2%
Comparisons	2	1.9	−5%

Results for random … use **pivot sampling**:

How to analyze that?



Java 6:
Ninther ← pseudomedian of 9



Java 7:
Tertiles of 5

Pivot Sampling

Setup: Sampling parameter $\vec{t} = (t_1, t_2, t_3)$

1. Sample $k = t_1 + t_2 + t_3 + 2$ elements
2. **Sort** the sample
3. Select pivots s. t. in sorted sample

   t_1 smaller,

   t_2 between and

   t_3 larger than pivots

$\rightsquigarrow$ tertiles of 5: $\vec{t} = (1, 1, 1)$
no pivot sampling: $\vec{t} = (0, 0, 0)$

- not necessarily symmetric:

  $\vec{t} = (3, 2, 1)$ $(k = 8)$:

Pivot Sampling

Setup: Sampling parameter $\vec{t} = (t_1, t_2, t_3)$

1. Sample $k = t_1 + t_2 + t_3 + 2$ elements
2. Sort the sample
3. Select pivots s. t. in sorted sample

   t_1 smaller,

   t_2 between and

   t_3 larger than pivots

$\rightsquigarrow$ tertiles of 5: $\vec{t} = (1, 1, 1)$
no pivot sampling: $\vec{t} = (0, 0, 0)$

- not necessarily symmetric:

  $\vec{t} = (3, 2, 1)$ ($k = 8$):

Pivot Sampling

Setup: Sampling parameter $\vec{t} = (t_1, t_2, t_3)$

1. Sample $k = t_1 + t_2 + t_3 + 2$ elements
2. **Sort** the sample
3. Select pivots s. t. in sorted sample

   t_1 smaller,
   t_2 between and
   t_3 larger than pivots

$\rightsquigarrow$ tertiles of 5: $\vec{t} = (1, 1, 1)$
no pivot sampling: $\vec{t} = (0, 0, 0)$

- not necessarily symmetric:

  $\vec{t} = (3, 2, 1)$ ($k = 8$):

Pivot Sampling

Setup: Sampling parameter $\vec{t} = (t_1, t_2, t_3)$

1. Sample $k = t_1 + t_2 + t_3 + 2$ elements
2. **Sort** the sample
3. Select pivots s. t. in sorted sample

   t_1 smaller,
   t_2 between and
   t_3 larger than pivots

⇝ tertiles of 5: $\vec{t} = (1, 1, 1)$
no pivot sampling: $\vec{t} = (0, 0, 0)$

- not necessarily symmetric:

  $\vec{t} = (3, 2, 1)$ $(k = 8)$:

Pivot Sampling

Setup: Sampling parameter $\vec{t} = (t_1, t_2, t_3)$

1. Sample $k = t_1 + t_2 + t_3 + 2$ elements
2. **Sort** the sample
3. Select pivots s. t. in sorted sample

   t_1 smaller,
   t_2 between and
   t_3 larger than pivots

$\rightsquigarrow$ tertiles of 5: $\vec{t} = (1, 1, 1)$
no pivot sampling: $\vec{t} = (0, 0, 0)$

- not necessarily symmetric:

  $\vec{t} = (3, 2, 1)$ $(k = 8)$:

Pivot Sampling

Setup: Sampling parameter $\vec{t} = (t_1, t_2, t_3)$

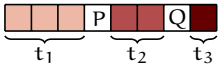
1. Sample $k = t_1 + t_2 + t_3 + 2$ elements
2. **Sort** the sample
3. Select pivots s. t. in sorted sample

   t_1 smaller,
   t_2 between and
   t_3 larger than pivots

⇝ tertiles of 5: $\vec{t} = (1, 1, 1)$
no pivot sampling: $\vec{t} = (0, 0, 0)$

- not necessarily symmetric:

  $\vec{t} = (3, 2, 1)$ ($k = 8$):



Pivot Sampling

Setup: Sampling parameter $\vec{t} = (t_1, t_2, t_3)$

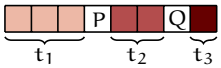
1. Sample $k = t_1 + t_2 + t_3 + 2$ elements
2. **Sort** the sample
3. Select pivots s. t. in sorted sample

   t_1 smaller,
   t_2 between and
   t_3 larger than pivots

$\rightsquigarrow$ tertiles of 5: $\vec{t} = (1, 1, 1)$
no pivot sampling: $\vec{t} = (0, 0, 0)$

- not necessarily symmetric:

  $\vec{t} = (3, 2, 1)$ ($k = 8$):



Analysis:

- Only distribution of $\vec{D}$ changes:
  $\vec{D} \overset{D}{=} \text{Dirichlet}(\vec{t} + 1)$

$\rightsquigarrow$ We still have

$$c(n) \sim \frac{2 - \mathbb{E}[D_3^2] - \mathbb{E}[D_1(D_1 + D_2)]}{\mathbb{E}[\mathcal{H}(\vec{D})]} \cdot n \ln n$$

Pivot Sampling

Setup: Sampling parameter $\vec{t} = (t_1, t_2, t_3)$

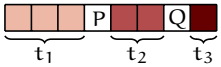
1. Sample $k = t_1 + t_2 + t_3 + 2$ elements
2. **Sort** the sample
3. Select pivots s. t. in sorted sample

   t_1 smaller,
   t_2 between and
   t_3 larger than pivots

$\rightsquigarrow$ tertiles of 5: $\vec{t} = (1, 1, 1)$
no pivot sampling: $\vec{t} = (0, 0, 0)$

- not necessarily symmetric:

  $\vec{t} = (3, 2, 1)$ ($k = 8$):



Analysis:

- Only distribution of $\vec{D}$ changes:
  $\vec{D} \overset{\mathcal{D}}{=} \text{Dirichlet}(\vec{t} + 1)$
  multivariate Beta distribution

$\rightsquigarrow$ We still have

$$c(n) \sim \frac{2 - \mathbb{E}[D_3^2] - \mathbb{E}[D_1(D_1 + D_2)]}{\mathbb{E}[\mathcal{H}(\vec{D})]} \cdot n \ln n$$

Pivot Sampling

Setup: Sampling parameter $\vec{t} = (t_1, t_2, t_3)$

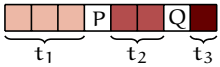
1. Sample $k = t_1 + t_2 + t_3 + 2$ elements
2. **Sort** the sample
3. Select pivots s. t. in sorted sample

   t_1 smaller,
   t_2 between and
   t_3 larger than pivots

$\rightsquigarrow$ tertiles of 5: $\vec{t} = (1, 1, 1)$
no pivot sampling: $\vec{t} = (0, 0, 0)$

- not necessarily symmetric:

  $\vec{t} = (3, 2, 1)$ $(k = 8)$:



Analysis:

- Only distribution of $\vec{D}$ changes:
  $\vec{D} \overset{\mathcal{D}}{=} \text{Dirichlet}(\vec{t} + 1)$
  multivariate Beta distribution

$\rightsquigarrow$ We still have

$$c(n) \sim \frac{2 - \mathbb{E}[D_3^2] - \mathbb{E}[D_1(D_1 + D_2)]}{\mathbb{E}[\mathcal{H}(\vec{D})]} \cdot n \ln n$$

Pivot Sampling

Setup: Sampling parameter $\vec{t} = (t_1, t_2, t_3)$

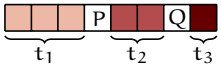
1. Sample $k = t_1 + t_2 + t_3 + 2$ elements
2. **Sort** the sample
3. Select pivots s. t. in sorted sample

   t_1 smaller,
   t_2 between and
   t_3 larger than pivots

$\rightsquigarrow$ tertiles of 5: $\vec{t} = (1, 1, 1)$
no pivot sampling: $\vec{t} = (0, 0, 0)$

- not necessarily symmetric:

  $\vec{t} = (3, 2, 1)$ $(k = 8)$:



Analysis:

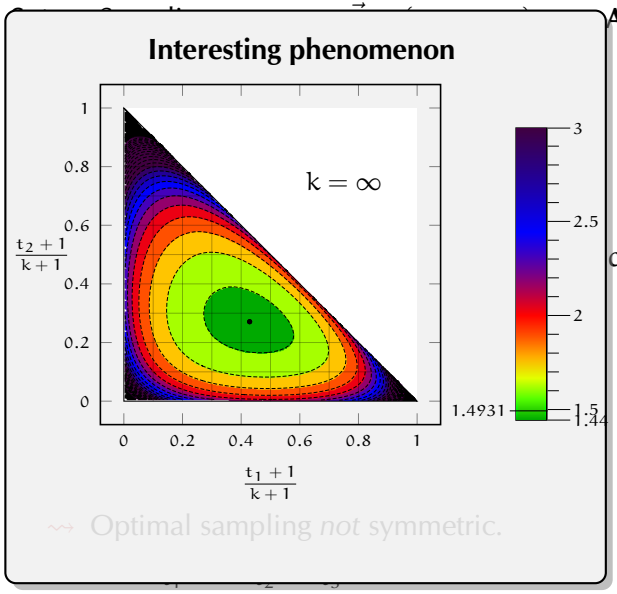
- Only distribution of $\vec{D}$ changes:
  $\vec{D} \overset{\mathcal{D}}{=} \text{Dirichlet}(\vec{t} + 1)$
  multivariate Beta distribution

$\rightsquigarrow$ We still have

$$c(n) \sim \frac{2 - \mathbb{E}[D_3^2] - \mathbb{E}[D_1(D_1 + D_2)]}{\mathbb{E}[\mathcal{H}(\vec{D})]} \cdot n \ln n$$

$$= \frac{1 + \frac{t_2+1}{k+1} + \frac{(2t_1+t_2+3)(t_3+1)}{(k+2)(k+1)}}{\sum_{r=1}^{3} \frac{t_r+1}{k+1}(H_{k+1} - H_{t_r+1})} \cdot n \ln n$$

Pivot Sampling



⇝ Optimal sampling *not* symmetric.

Analysis:

- Only distribution of $\vec{D}$ changes:
  $\vec{D} \overset{\mathcal{D}}{=} \text{Dirichlet}(\vec{t}+1)$
  ↖ multivariate Beta distribution

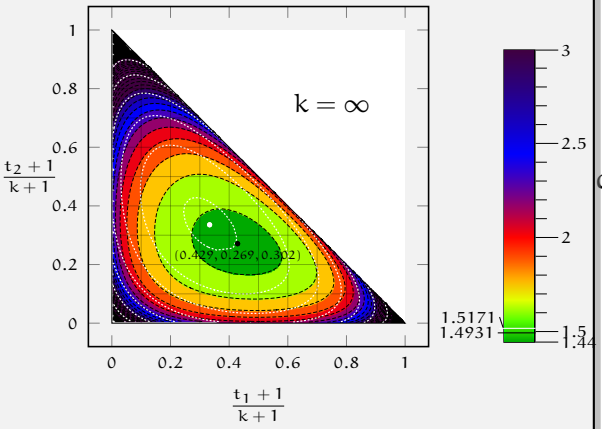
⇝ We still have

$$c(n) \sim \frac{2-\mathbb{E}[D_3^2]-\mathbb{E}[D_1(D_1+D_2)]}{\mathbb{E}[\mathcal{H}(\vec{D})]} \cdot n \ln n$$

$$= \frac{1+\frac{t_2+1}{k+1}+\frac{(2t_1+t_2+3)(t_3+1)}{(k+2)(k+1)}}{\sum_{r=1}^{3}\frac{t_r+1}{k+1}(H_{k+1}-H_{t_r+1})} \cdot n \ln n$$

Pivot Sampling

Interesting phenomenon



⇝ Optimal sampling *not* symmetric.

Analysis:

- Only distribution of $\vec{D}$ changes:
  $\vec{D} \overset{\mathcal{D}}{=} \text{Dirichlet}(\vec{t}+1)$
  ↖ multivariate Beta distribution

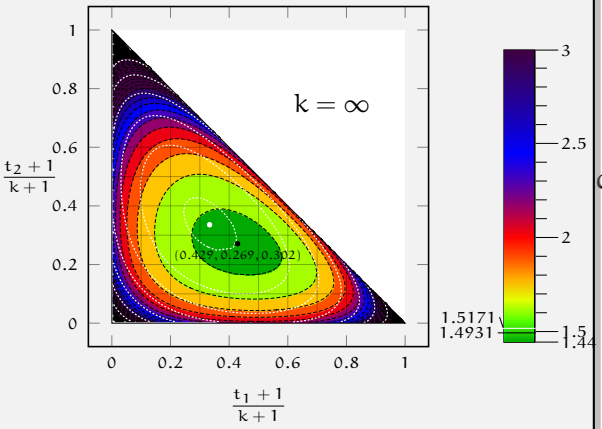
⇝ We still have

$$c(n) \sim \frac{2-\mathbb{E}[D_3^2]-\mathbb{E}[D_1(D_1+D_2)]}{\mathbb{E}[\mathcal{H}(\vec{D})]} \cdot n \ln n$$

$$= \frac{1+\frac{t_2+1}{k+1}+\frac{(2t_1+t_2+3)(t_3+1)}{(k+2)(k+1)}}{\sum_{r=1}^{3}\frac{t_r+1}{k+1}(H_{k+1}-H_{t_r+1})} \cdot n \ln n$$

Pivot Sampling

Interesting phenomenon



⇝ Optimal sampling *not* symmetric.

Analysis:

- Only distribution of $\vec{D}$ changes:
  $\vec{D} \overset{\mathcal{D}}{=} \text{Dirichlet}(\vec{t}+1)$
  multivariate Beta distribution

⇝ We still have

$$c(n) \sim \frac{2-\mathbb{E}[D_3^2]-\mathbb{E}[D_1(D_1+D_2)]}{\mathbb{E}[\mathcal{H}(\vec{D})]} \cdot n \ln n$$

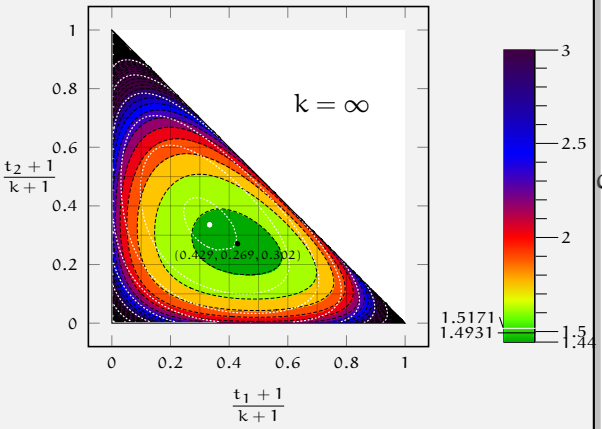
$$= \frac{1+\frac{t_2+1}{k+1}+\frac{(2t_1+t_2+3)(t_3+1)}{(k+2)(k+1)}}{\sum_{r=1}^{3}\frac{t_r+1}{k+1}(H_{k+1}-H_{t_r+1})} \cdot n \ln n$$

- Java 7: $\vec{t} = (1,1,1)$

  $c(n) \sim \frac{680}{399} n \ln n \approx 1.70426\, n \ln n$

Pivot Sampling

Interesting phenomenon



⇝ Optimal sampling *not* symmetric.

Analysis:

- Only distribution of $\vec{D}$ changes:
  $\vec{D} \overset{\mathcal{D}}{=} \text{Dirichlet}(\vec{t}+1)$
  multivariate Beta distribution

⇝ We still have

$$c(n) \sim \frac{2-\mathbb{E}[D_3^2]-\mathbb{E}[D_1(D_1+D_2)]}{\mathbb{E}[\mathcal{H}(\vec{D})]} \cdot n \ln n$$

$$= \frac{1+\frac{t_2+1}{k+1}+\frac{(2t_1+t_2+3)(t_3+1)}{(k+2)(k+1)}}{\sum_{r=1}^{3}\frac{t_r+1}{k+1}(H_{k+1}-H_{t_r+1})} \cdot n \ln n$$

- Java 7: $\vec{t} = (1,1,1)$

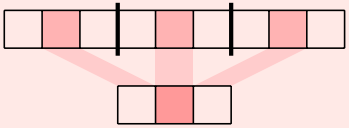
  $c(n) \sim \frac{680}{399} n \ln n \approx 1.70426\, n \ln n$

Classic vs. YBB Quicksort

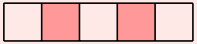
- Time for the competition!

	CQS	YQS	Relative
Running Time	(from various experiments)		$-10\pm2\%$
Comparisons	2	1.9	-5%

Results for random pivots ... but Java 6/7 use **pivot sampling**:



Java 6:
Ninther



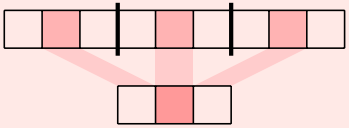
Java 7:
Tertiles of 5

Classic vs. YBB Quicksort

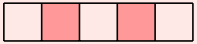
- Time for the competition!

	CQS	YQS	Relative
Running Time	(from various experiments)		$-10\pm2\%$
Comparisons	2	1.9	-5%
(with sampling as in JRE)		1.70	

Results for random pivots ... but Java 6/7 use **pivot sampling**:



Java 6:
Ninther



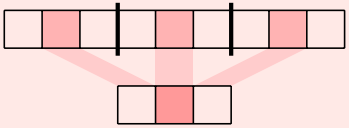
Java 7:
Tertiles of 5

Classic vs. YBB Quicksort

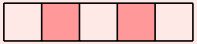
- Time for the competition!

	CQS	YQS	Relative
Running Time	(from various experiments)		−10±2%
Comparisons	2	1.9	−5%
(with sampling as in JRE)	1.57	1.70	+8.6%

Results for random pivots ... but Java 6/7 use **pivot sampling**:



Java 6:
Ninther



Java 7:
Tertiles of 5

Classic vs. YBB Quicksort

- Time for the competition!

	CQS	YQS	Relative
Running Time	(from various experiments)		−10±2%
Comparisons	2	1.9	−5%
(with sampling as in JRE)	1.57	1.70	+8.6%
Swaps	$0.\overline{3}$	0.6	+80%

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

Classic vs. YBB Quicksort

- Time for the competition!

	CQS	YQS	Relative
Running Time	(from various experiments)		$-10\pm2\%$
Comparisons	2	1.9	-5%
(with sampling as in JRE)	1.57	1.70	$+8.6\%$
Swaps	$0.3\overline{3}$	0.6	$+80\%$
Bytecode Instructions	18	21.7	$+20.6\%$
MMIX oops υ	11	13.1	$+19.1\%$
MMIX mems μ	$2.\overline{6}$	2.8	$+5\%$

Knuth's mythical machine → MMIX

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

Classic vs. YBB Quicksort

- Time for the competition!

	CQS	YQS	Relative
Running Time	(from various experiments)		−10±2%
Comparisons	2	1.9	−5%
(with sampling as in JRE)	1.57	1.70	+8.6%
Swaps	$0.\overline{3}$	0.6	+80%
Bytecode Instructions	18	21.7	+20.6%
MMIX oops υ	11	13.1	+19.1%
MMIX mems μ	$2.\overline{6}$	2.8	+5%
Branch Mispredictions	0.57	0.58	+2%

Knuth's mythical machine → MMIX

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

The *Memory Wall*

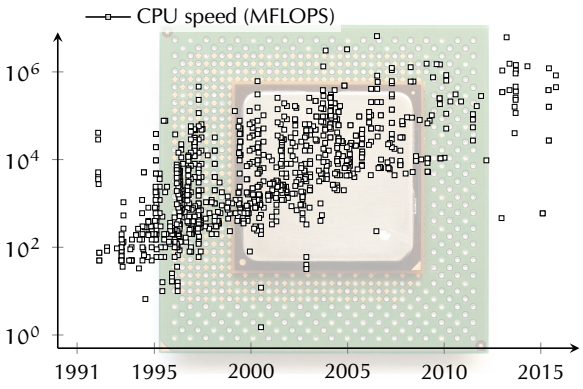
Comparisons $\widehat{=}$ **CPU** effort, but we also need to **load** items from **memory**.

The *Memory Wall*

Comparisons $\widehat{=}$ **CPU** effort, but we also need to **load** items from **memory**.

The *Memory Wall*

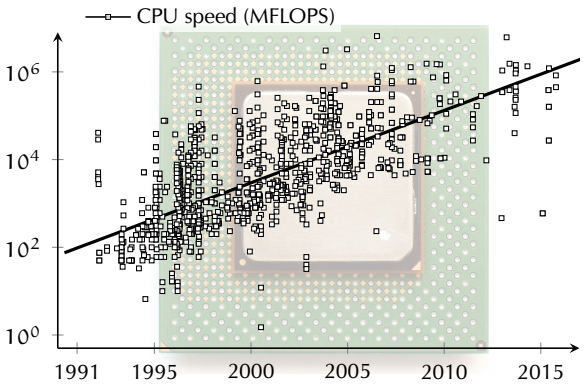
Comparisons $\widehat{=}$ **CPU** effort, but we also need to **load** items from **memory**.



STREAM benchmark data with linear regressions
`www.cs.virginia.edu/stream/by_date/Balance.html`

The *Memory Wall*

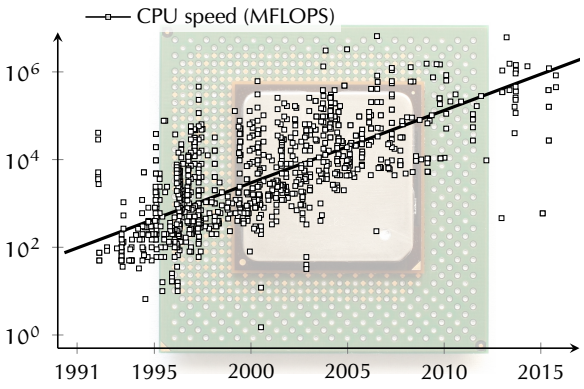
Comparisons $\widehat{=}$ **CPU** effort, but we also need to **load** items from **memory**.



STREAM benchmark data with linear regressions
`www.cs.virginia.edu/stream/by_date/Balance.html`

The *Memory Wall*

Comparisons $\widehat{=}$ **CPU** effort, but we also need to **load** items from **memory**.



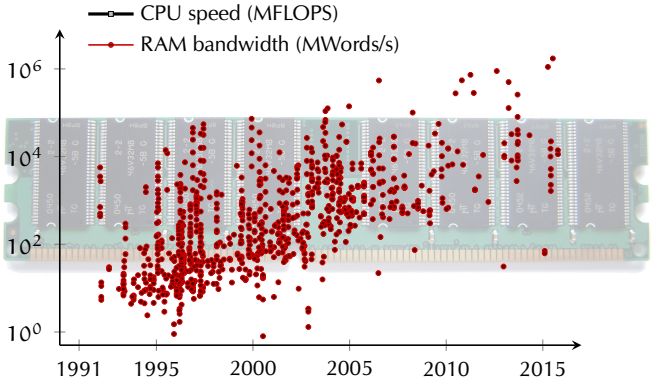
STREAM benchmark data with linear regressions
`www.cs.virginia.edu/stream/by_date/Balance.html`

Averaged annual growth rates:

Processor speed: 46%

The *Memory Wall*

Comparisons $\widehat{=}$ **CPU** effort, but we also need to **load** items from **memory**.



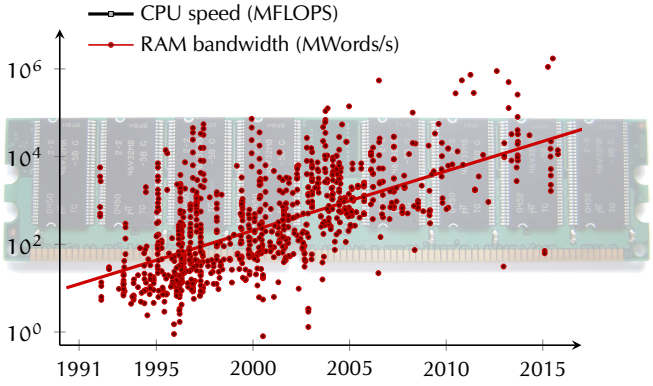
Averaged annual growth rates:

Processor speed: 46%

STREAM benchmark data with linear regressions
`www.cs.virginia.edu/stream/by_date/Balance.html`

The *Memory Wall*

Comparisons $\widehat{=}$ **CPU** effort, but we also need to **load** items from **memory**.



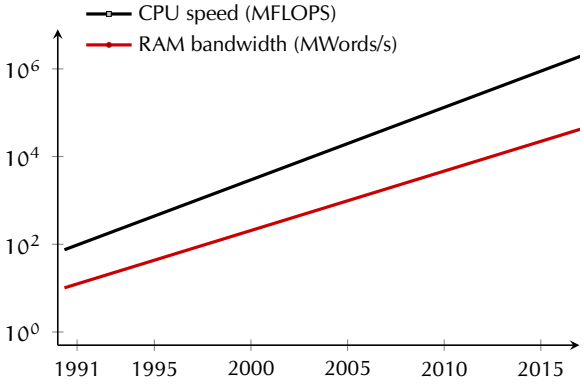
Averaged annual growth rates:

Processor speed: 46%
Memory Bandwidth: 37%

STREAM benchmark data with linear regressions
`www.cs.virginia.edu/stream/by_date/Balance.html`

The *Memory Wall*

Comparisons $\widehat{=}$ **CPU** effort, but we also need to **load** items from **memory**.



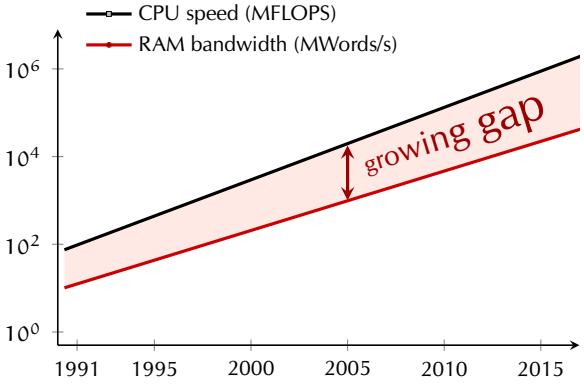
Averaged annual growth rates:

Processor speed: 46%
Memory Bandwidth: 37%

STREAM benchmark data with linear regressions
`www.cs.virginia.edu/stream/by_date/Balance.html`

The *Memory Wall*

Comparisons $\widehat{=}$ **CPU** effort, but we also need to **load** items from **memory**.



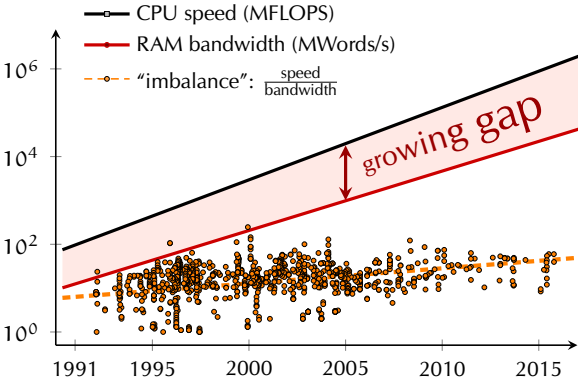
Averaged annual growth rates:

Processor speed: 46%
Memory Bandwidth: 37%

STREAM benchmark data with linear regressions
`www.cs.virginia.edu/stream/by_date/Balance.html`

The *Memory Wall*

Comparisons $\widehat{=}$ **CPU** effort, but we also need to **load** items from **memory**.



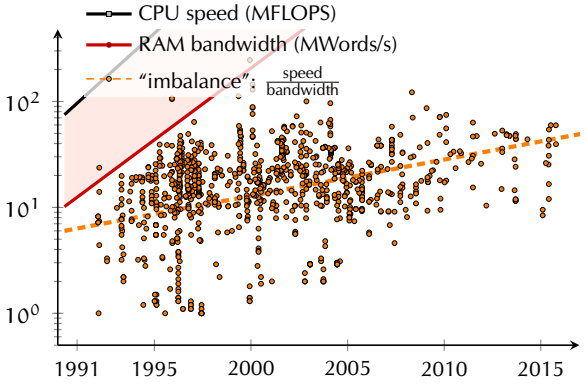
STREAM benchmark data with linear regressions
`www.cs.virginia.edu/stream/by_date/Balance.html`

Averaged annual growth rates:

Processor speed: 46%
Memory Bandwidth: 37%

The *Memory Wall*

Comparisons $\widehat{=}$ **CPU** effort, but we also need to **load** items from **memory**.



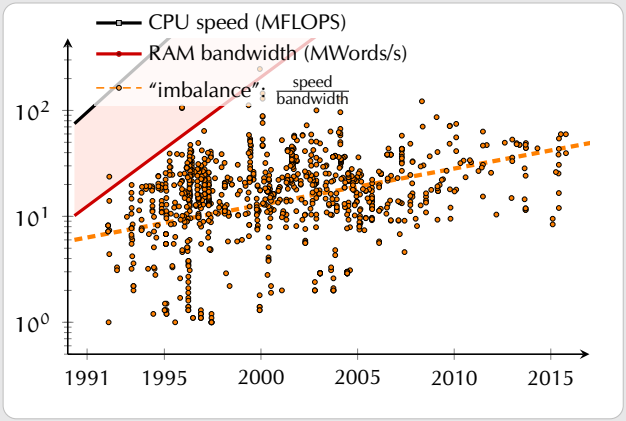
STREAM benchmark data with linear regressions
`www.cs.virginia.edu/stream/by_date/Balance.html`

Averaged annual growth rates:

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

The *Memory Wall*

Comparisons $\widehat{=}$ **CPU** effort, but we also need to **load** items from **memory**.



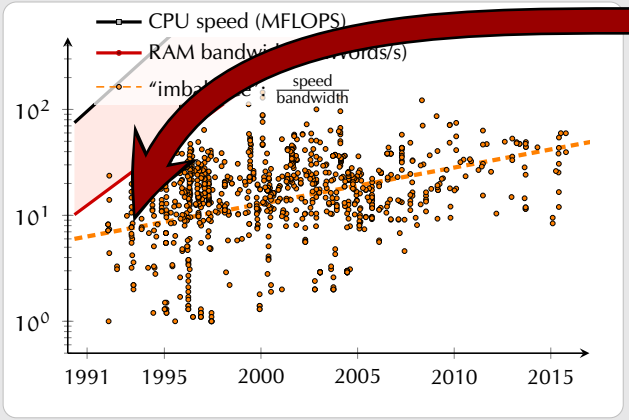
STREAM benchmark data with linear regressions
www.cs.virginia.edu/stream/by_date/Balance.html

Averaged annual growth rates:

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

The *Memory Wall*

Comparisons $\widehat{=}$ **CPU** effort, but we also need



STREAM benchmark data with linear regressions
www.cs.virginia.edu/stream/by_date/Balance.html

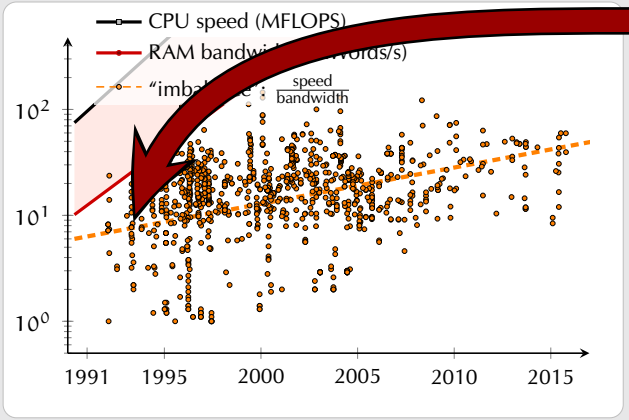
Classic Quicksort (Bentley & McIlroy) **1993**

- 1 mem. transfer $\approx$ 7 operations
- $\rightsquigarrow$ optimized for CPU

Memory Bandwidth: 37%
$\rightsquigarrow$ Imbalance: 5%

The *Memory Wall*

Comparisons $\widehat{=}$ **CPU** effort, but we also need



STREAM benchmark data with linear regressions
www.cs.virginia.edu/stream/by_date/Balance.html

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

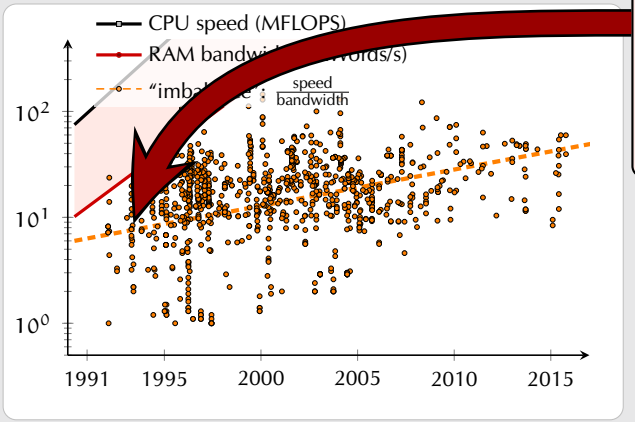
- 1 mem. transfer $\approx$ 7 operations

$\rightsquigarrow$ optimized for CPU

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

The *Memory Wall*

Comparisons $\widehat{=}$ **CPU** effort, but we also need

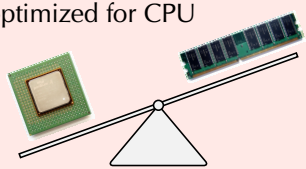


STREAM benchmark data with linear regressions
www.cs.virginia.edu/stream/by_date/Balance.html

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

- 1 mem. transfer $\approx$ 7 operations

$\rightsquigarrow$ optimized for CPU

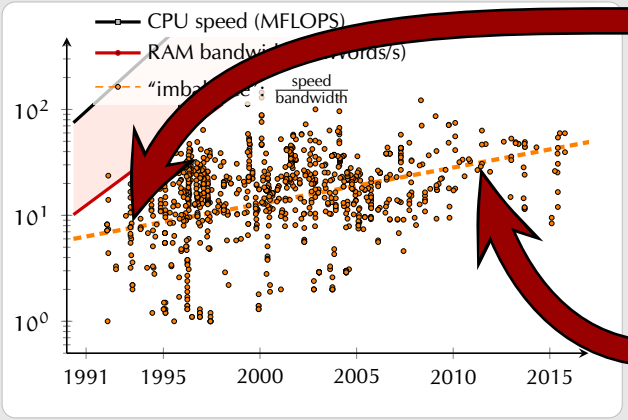


Memory Bandwidth: 37%

$\rightsquigarrow$ Imbalance: 5%

The *Memory Wall*

Comparisons $\widehat{=}$ **CPU** effort, but we also need

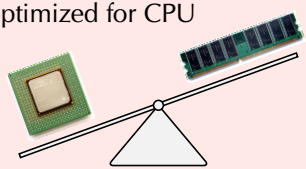


STREAM benchmark data with linear regressions
www.cs.virginia.edu/stream/by_date/Balance.html

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

- 1 mem. transfer $\approx$ 7 operations

$\rightsquigarrow$ optimized for CPU



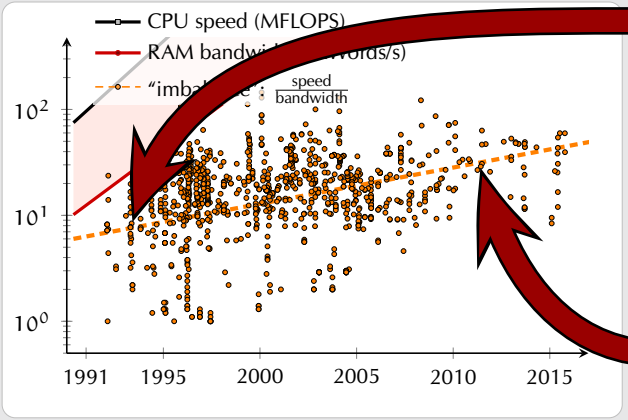
Dual-Pivot Quicksort, **2009 – today**

- 1 mem. transfer $\approx$ 30 operations

$\rightsquigarrow$ optimize for CPU **and bandwidth?**

The *Memory Wall*

Comparisons $\widehat{=}$ **CPU** effort, but we also need

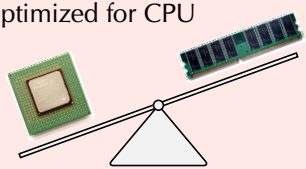


STREAM benchmark data with linear regressions
www.cs.virginia.edu/stream/by_date/Balance.html

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

- 1 mem. transfer $\approx$ 7 operations

$\rightsquigarrow$ optimized for CPU



Processor speed: 40%

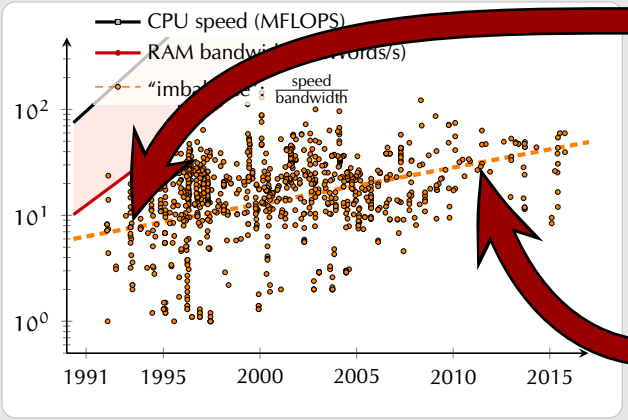
Dual-Pivot Quicksort, **2009 – today**

- 1 mem. transfer $\approx$ 30 operations

$\rightsquigarrow$ optimize for CPU and bandwidth?

The *Memory Wall*

Comparisons $\widehat{=}$ **CPU** effort, but we also need

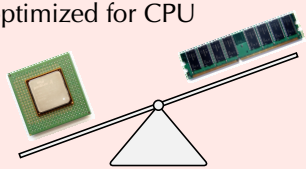


STREAM benchmark data with linear regressions
www.cs.virginia.edu/stream/by_date/Balance.html

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

- 1 mem. transfer $\approx$ 7 operations

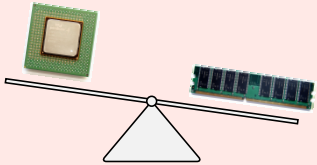
$\rightsquigarrow$ optimized for CPU



Dual-Pivot Quicksort, **2009 – today**

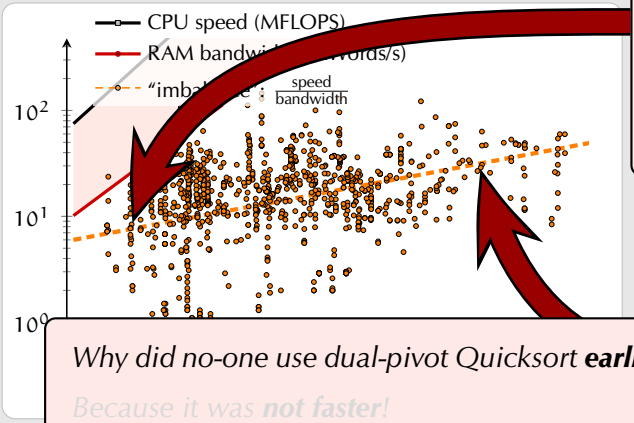
- 1 mem. transfer $\approx$ 30 operations

$\rightsquigarrow$ optimize for CPU **and bandwidth**?



The *Memory Wall*

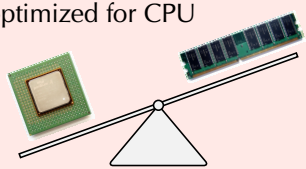
Comparisons $\widehat{=}$ **CPU** effort, but we also need



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

- 1 mem. transfer $\approx$ 7 operations

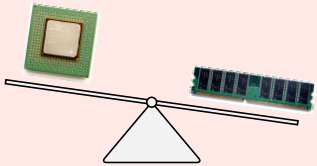
$\rightsquigarrow$ optimized for CPU



Dual-Pivot Quicksort, **2009 – today**

- 1 mem. transfer $\approx$ 30 operations

$\rightsquigarrow$ optimize for CPU **and bandwidth**?



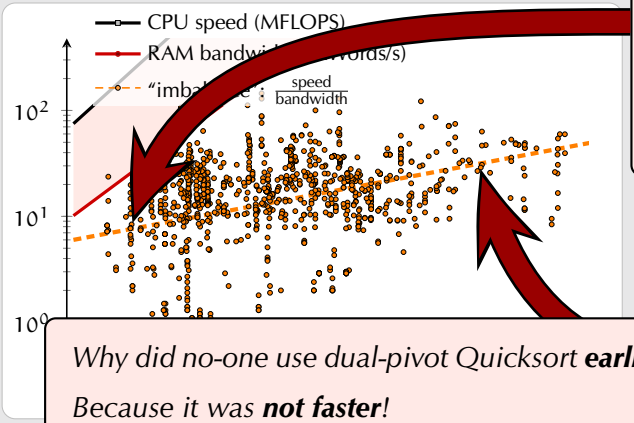
*Why did no-one use dual-pivot Quicksort **earlier**?*

*Because it was **not faster**!*

STREAM benchmark data with linear regressions
www.cs.virginia.edu/stream/by_date/Balance.html

The *Memory Wall*

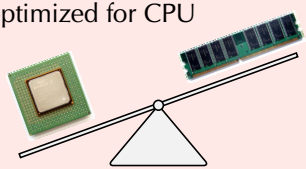
Comparisons $\widehat{=}$ **CPU** effort, but we also need



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

- 1 mem. transfer $\approx$ 7 operations

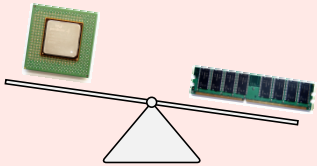
$\rightsquigarrow$ optimized for CPU



Dual-Pivot Quicksort, **2009 – today**

- 1 mem. transfer $\approx$ 30 operations

$\rightsquigarrow$ optimize for CPU **and bandwidth**?



*Why did no-one use dual-pivot Quicksort **earlier**?*

*Because it was **not faster**!*

STREAM benchmark data with linear regressions
www.cs.virginia.edu/stream/by_date/Balance.html

Scanned Elements

How to analyze bandwidth consumption?

Need good cost measure!

Scanned Elements

How to analyze bandwidth consumption?

Need good cost measure!

Scanned Elements

How to analyze bandwidth consumption?

Need good cost measure!

Why reinvent the wheel?

- *I/Os* in *external-memory model*
  - additional parameters M and B
  - typically: $O(\cdot)$, **only** I/Os count
- *cache misses*
  - hardware-specific
  - misses vs. bandwidth

Scanned Elements

How to analyze bandwidth consumption?

Need good cost measure!

Why reinvent the wheel?

- ***I/Os* in *external-memory model***
  - additional parameters M and B
  - typically: O(○), **only** I/Os count
- ***cache misses***
  - hardware-specific
  - misses vs. bandwidth

Scanned Elements

How to analyze bandwidth consumption?

Need good cost measure!

Why reinvent the wheel?

- ***I/Os* in *external-memory model***
  - additional parameters M and B
  - typically: $O(\circ)$, **only** I/Os count
- ***cache misses***
  - hardware-specific
  - misses vs. bandwidth

Scanned Elements

How to analyze bandwidth consumption?

Need good cost measure!

Why reinvent the wheel?

- ***I/Os* in *external-memory model***
  - additional parameters M and B
  - typically: $O(\circ)$, **only** I/Os count
- ***cache misses***
  - hardware-specific
  - misses vs. bandwidth

Scanned Elements

How to analyze bandwidth consumption?

Need good cost measure!

Why reinvent the wheel?

- ***I/Os* in *external-memory model***
  - additional parameters M and B
  - typically: $O(\circ)$, **only** I/Os count
- ***cache misses***
  - hardware-specific
  - misses vs. bandwidth

Scanned Elements

How to analyze bandwidth consumption?

Need good cost measure!

Why reinvent the wheel?

- ***I/Os* in *external-memory model***
  - additional parameters M and B
  - typically: $O(\circ)$, **only** I/Os count
- ***cache misses***
  - hardware-specific
  - misses vs. bandwidth

Scanned Elements

How to analyze bandwidth consumption?

Need good cost measure!

Why reinvent the wheel?

- ***I/Os* in *external-memory model***
  - additional parameters M and B
  - typically: $O(\circ)$, **only** I/Os count
- ***cache misses***
  - hardware-specific
  - misses vs. bandwidth

Scanned Elements

How to analyze bandwidth consumption?

Need good cost measure!

Proposal: count ***scanned elements***

- Access to array only through **iterators**
  (no random access!)
- Iterators can
  - head left or right (one-directional!)
  - **advance** to next position
  - **read** and **write** current cell
- Cost: number of **advances**

≈ **cache misses**, but hardware independent

Why reinvent the wheel?

- ***I/Os* in *external-memory model***
  - additional parameters M and B
  - typically: $O(\circ)$, **only** I/Os count
- ***cache misses***
  - hardware-specific
  - misses vs. bandwidth

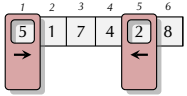
Scanned Elements

How to analyze bandwidth consumption?

Need good cost measure!

Proposal: count ***scanned elements***

- Access to array only through **iterators**
  (no random access!)



- Iterators can
  - head left or right (one-directional!)
  - **advance** to next position
  - **read** and **write** current cell
- Cost: number of **advances**

≈ **cache misses**, but hardware independent

Why reinvent the wheel?

- ***I/Os* in *external-memory model***
  - additional parameters M and B
  - typically: $O(\circ)$, **only** I/Os count
- ***cache misses***
  - hardware-specific
  - misses vs. bandwidth

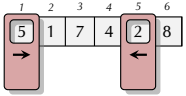
Scanned Elements

How to analyze bandwidth consumption?

Need good cost measure!

Proposal: count ***scanned elements***

- Access to array only through **iterators**
  (no random access!)



- Iterators can
  - head left or right (one-directional!)
  - **advance** to next position
  - **read** and **write** current cell
- Cost: number of **advances**

≈ **cache misses**, but hardware independent

Why reinvent the wheel?

- ***I/Os* in *external-memory model***
  - additional parameters M and B
  - typically: $O(\circ)$, **only** I/Os count
- ***cache misses***
  - hardware-specific
  - misses vs. bandwidth

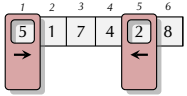
Scanned Elements

How to analyze bandwidth consumption?

Need good cost measure!

Proposal: count ***scanned elements***

- Access to array only through **iterators**
  (no random access!)



- Iterators can
  - head left or right (one-directional!)
  - **advance** to next position
  - **read** and **write** current cell
- Cost: number of **advances**

≈ **cache misses**, but hardware independent

Why reinvent the wheel?

- ***I/Os* in *external-memory model***
  - additional parameters M and B
  - typically: $O(\circ)$, **only** I/Os count
- ***cache misses***
  - hardware-specific
  - misses vs. bandwidth

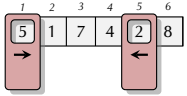
Scanned Elements

How to analyze bandwidth consumption?

Need good cost measure!

Proposal: count ***scanned elements***

- Access to array only through **iterators**
  (no random access!)



- Iterators can
  - head left or right (one-directional!)
  - **advance** to next position
  - **read** and **write** current cell
- Cost: number of **advances**

≈ **cache misses**, but hardware independent

Why reinvent the wheel?

- ***I/Os* in *external-memory model***
  - additional parameters M and B
  - typically: $O(\circ)$, **only** I/Os count
- ***cache misses***
  - hardware-specific
  - misses vs. bandwidth

Scanned Elements

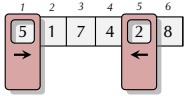
How to analyze bandwidth consumption?

Need good cost measure!

Proposal: count ***scanned elements***

- Access to array only through **iterators**
  (no random access!)
- Iterators can
  - head left or right (one-directional!)
  - **advance** to next position
  - **read** and **write** current cell
- Cost: number of **advances**

≈ **cache misses**, but hardware independent



Why reinvent the wheel?

- ***I/Os* in *external-memory model***
  - additional parameters M and B
  - typically: $O(\circ)$, **only** I/Os count
- ***cache misses***
  - hardware-specific
  - misses vs. bandwidth

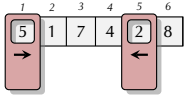
Scanned Elements

How to analyze bandwidth consumption?

Need good cost measure!

Proposal: count ***scanned elements***

- Access to array only through **iterators**
  (no random access!)



- Iterators can
  - head left or right (one-directional!)
  - **advance** to next position
  - **read** and **write** current cell
- Cost: number of **advances**

≈ **cache misses**, but hardware independent

Why reinvent the wheel?

- ***I/Os* in *external-memory model***
  - additional parameters M and B
  - typically: $O(\circ)$, **only** I/Os count
- ***cache misses***
  - hardware-specific
  - misses vs. bandwidth

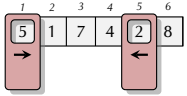
Scanned Elements

How to analyze bandwidth consumption?

Need good cost measure!

Proposal: count ***scanned elements***

- Access to array only through **iterators**
  (no random access!)



- Iterators can
  - head left or right (one-directional!)
  - **advance** to next position
  - **read** and **write** current cell
- Cost: number of **advances**

≈ **cache misses**, but hardware independent

Why reinvent the wheel?

- ***I/Os* in *external-memory model***
  - additional parameters M and B
  - typically: $O(\circ)$, **only** I/Os count
- ***cache misses***
  - hardware-specific
  - misses vs. bandwidth

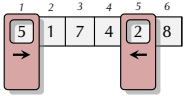
Scanned Elements

How to analyze bandwidth consumption?

Need good cost measure!

Proposal: count ***scanned elements***

- Access to array only through **iterators**
  (no random access!)



- Iterators can
  - head left or right (one-directional!)
  - **advance** to next position
  - **read** and **write** current cell
- Cost: number of **advances**

≈ **cache misses**, but hardware independent

Costs of Partitioning

Classic Quicksort

⇝ n scanned elements
(array scanned once)

YBB Quicksort

⇝ $1.\overline{3}n$ scanned elements
(on average without sampling)

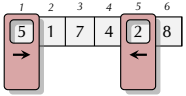
Scanned Elements

How to analyze bandwidth consumption?

Need good cost measure!

Proposal: count ***scanned elements***

- Access to array only through **iterators**
  (no random access!)

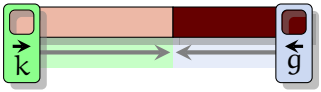


- Iterators can
  - head left or right (one-directional!)
  - **advance** to next position
  - **read** and **write** current cell
- Cost: number of **advances**

$\approx$ **cache misses**, but hardware independent

Costs of Partitioning

Classic Quicksort



$\rightsquigarrow$ n scanned elements
(array scanned once)

YBB Quicksort

$\rightsquigarrow$ $1.\overline{3}n$ scanned elements
(on average without sampling)

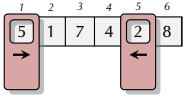
Scanned Elements

How to analyze bandwidth consumption?

Need good cost measure!

Proposal: count ***scanned elements***

- Access to array only through **iterators**
  (no random access!)

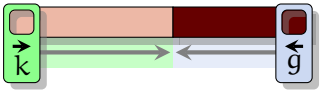


- Iterators can
  - head left or right (one-directional!)
  - **advance** to next position
  - **read** and **write** current cell
- Cost: number of **advances**

$\approx$ **cache misses**, but hardware independent

Costs of Partitioning

Classic Quicksort



$\rightsquigarrow$ n scanned elements
(array scanned once)

YBB Quicksort

$\rightsquigarrow$ $1.\overline{3}n$ scanned elements
(on average without sampling)

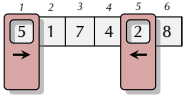
Scanned Elements

How to analyze bandwidth consumption?

Need good cost measure!

Proposal: count ***scanned elements***

- Access to array only through **iterators**
  (no random access!)

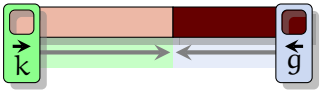


- Iterators can
  - head left or right (one-directional!)
  - **advance** to next position
  - **read** and **write** current cell
- Cost: number of **advances**

$\approx$ **cache misses**, but hardware independent

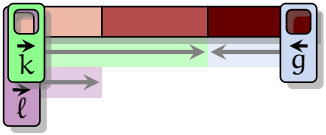
Costs of Partitioning

Classic Quicksort



$\rightsquigarrow$ n scanned elements
(array scanned once)

YBB Quicksort



$\rightsquigarrow$ $1.\bar{3}n$ scanned elements
(on average without sampling)

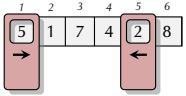
Scanned Elements

How to analyze bandwidth consumption?

Need good cost measure!

Proposal: count ***scanned elements***

- Access to array only through **iterators**
  (no random access!)

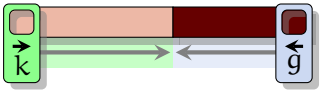


- Iterators can
  - head left or right (one-directional!)
  - **advance** to next position
  - **read** and **write** current cell
- Cost: number of **advances**

$\approx$ **cache misses**, but hardware independent

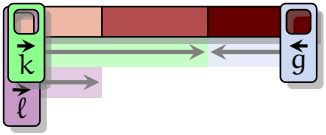
Costs of Partitioning

Classic Quicksort



$\rightsquigarrow$ n scanned elements
(array scanned once)

YBB Quicksort



$\rightsquigarrow$ $1.\overline{3}n$ scanned elements
(on average without sampling)

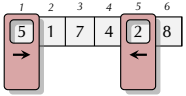
Scanned Elements

How to analyze bandwidth consumption?

Need good cost measure!

Proposal: count ***scanned elements***

- Access to array only through **iterators**
  (no random access!)

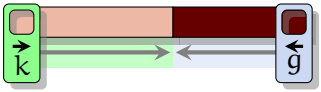


- Iterators can
  - head left or right (one-directional!)
  - **advance** to next position
  - **read** and **write** current cell
- Cost: number of **advances**

$\approx$ **cache misses**, but hardware independent

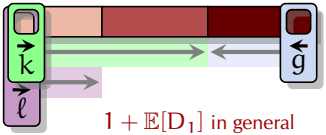
Costs of Partitioning

Classic Quicksort



$\rightsquigarrow$ n scanned elements
(array scanned once)

YBB Quicksort



$\rightsquigarrow$ $1.\overline{3}n$ scanned elements
(on average without sampling)

Classic vs. YBB Quicksort

- Time for the competition!

	CQS	YQS	Relative
Running Time	(from various experiments)		$-10\pm2\%$
Comparisons	2	1.9	-5%
(with sampling as in JRE)	1.57	1.70	$+8.6\%$
Swaps	$0.\overline{3}$	0.6	$+80\%$
Bytecode Instructions	18	21.7	$+20.6\%$
MMIX oops υ	11	13.1	$+19.1\%$
MMIX mems μ	$2.\overline{6}$	2.8	$+5\%$
Branch Mispredictions	0.57	0.58	$+2\%$

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

Classic vs. YBB Quicksort

- Time for the competition!

	CQS	YQS	Relative
Running Time	(from various experiments)		−10±2%
Comparisons	2	1.9	−5%
(with sampling as in JRE)	1.57	1.70	+8.6%
Swaps	$0.\overline{3}$	0.6	+80%
Bytecode Instructions	18	21.7	+20.6%
MMIX oops υ	11	13.1	+19.1%
MMIX mems μ	$2.\overline{6}$	2.8	+5%
Branch Mispredictions	0.57	0.58	+2%
Scanned Elements	2	1.6	−20%

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

Classic vs. YBB Quicksort

- Time for the competition!

	CQS	YQS	Relative
Running Time	(from various experiments)		−10±2%
Comparisons	2	1.9	−5%
(with sampling as in JRE)	1.57	1.70	+8.6%
Swaps	$0.\overline{3}$	0.6	+80%
Bytecode Instructions	18	21.7	+20.6%
MMIX oops υ	11	13.1	+19.1%
MMIX mems μ	$2.\overline{6}$	2.8	+5%
Branch Mispredictions	0.57	0.58	+2%
Scanned Elements	2	1.6	−20%

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

Only plausible explanation for running time: 20% less memory transfers in YQS.

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: $\vec{t} \in \mathbb{N}_0^s$

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: $\vec{t} \in \mathbb{N}_0^s$

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: $\vec{t} \in \mathbb{N}_0^s$

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

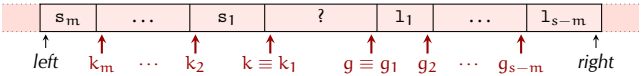
- classify elements using BSTs: λ_k and λ_g
- pivots $P_1, \ldots, P_{s-1}$ chosen from sample: $\vec{t} \in \mathbb{N}_0^s$

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



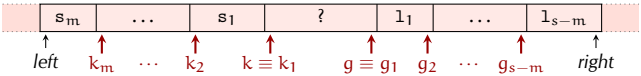
- classify elements using BSTs: λ_k and λ_g
- pivots $P_1, \ldots, P_{s-1}$ chosen from sample: $\vec{t} \in \mathbb{N}_0^s$

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



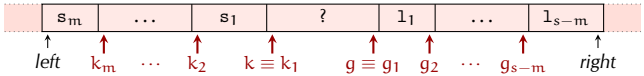
- classify elements using BSTs: λ_k and λ_g
- pivots $P_1, \ldots, P_{s-1}$ chosen from sample: $\vec{t} \in \mathbb{N}_0^s$

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 $P_1, \dots, P_{s-1}$ chosen from sample: $\vec{t} \in \mathbb{N}_0^s$

Quicksort variants as special cases

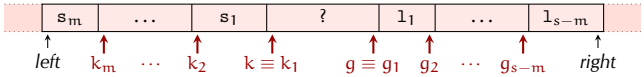
Method	s	m	λ_k	λ_g

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: $\vec{t} \in \mathbb{N}_0^s$

Quicksort variants as special cases

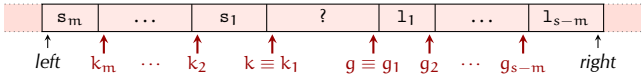
Method	s	m	λ_k	λ_g
Classic	2	1		

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: $\vec{t} \in \mathbb{N}_0^s$

Quicksort variants as special cases

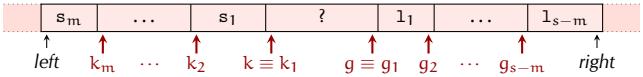
Method	s	m	λ_k	λ_g
Classic	2	1		
Lomuto	2	2		

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: $\vec{t} \in \mathbb{N}_0^s$

Quicksort variants as special cases

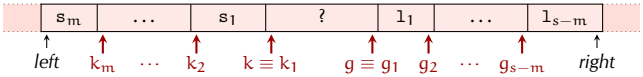
Method	s	m	λ_k	λ_g
Classic	2	1		
Lomuto	2	2		
YBB	3	2		

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: $\vec{t} \in \mathbb{N}_0^s$

Quicksort variants as special cases

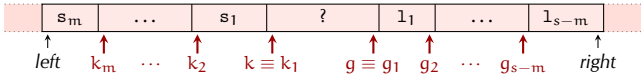
Method	s	m	λ_k	λ_g
Classic	2	1		
Lomuto	2	2		
YBB	3	2		
Waterloo	4	2		

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: $\vec{t} \in \mathbb{N}_0^s$

Quicksort variants as special cases

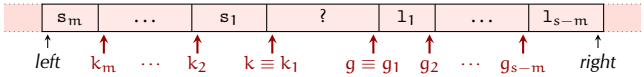
Method	s	m	λ_k	λ_g
Classic	2	1		
Lomuto	2	2		
YBB	3	2		
Waterloo	4	2		

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 $P_1, \dots, P_{s-1}$ chosen from sample: $\vec{t} \in \mathbb{N}_0^s$

Quicksort variants as special cases

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

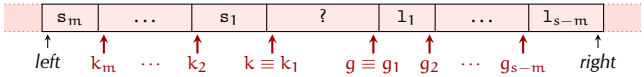
Same analysis techniques apply: $c(n) \sim \dfrac{\mathbb{E}[\#\text{cmps to classify 1 element}]}{\mathbb{E}\left[\mathcal{H}(\vec{D})\right]} \cdot n \ln n$

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 $P_1, \dots, P_{s-1}$ chosen from sample: $\vec{t} \in \mathbb{N}_0^s$

Quicksort variants as special cases

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

Same analysis techniques apply: $c(n) \sim \dfrac{\mathbb{E}[\#\text{cmps to classify 1 element}]}{\mathbb{E}\left[\mathcal{H}(\vec{D})\right]} \cdot n \ln n$

Fair Competition?

Consider the #comparisons again.

4-way Quicksort

- Always need 2 cmps to classify.

$\rightsquigarrow \frac{24}{13} n \ln(n)$ cmps (without sampling)

≈ 1.846

Classic Quicksort

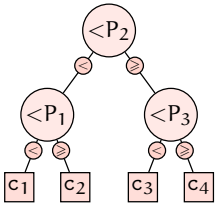
- $2 n \ln(n)$ comparisons (without sampling)

Fair Competition?

Consider the #comparisons again.

situation for YBB similar

4-way Quicksort



- Always need 2 cmps to classify.

$\rightsquigarrow$ $\frac{24}{13} n \ln(n)$ cmps (without sampling)

≈ 1.846

Classic Quicksort

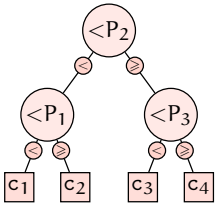
- $2 n \ln(n)$ comparisons (without sampling)

Fair Competition?

Consider the #comparisons again.

situation for YBB similar

4-way Quicksort



- Always need 2 cmps to classify.

$\rightsquigarrow$ $\frac{24}{13} n \ln(n)$ cmps (without sampling)

≈ 1.846

Classic Quicksort

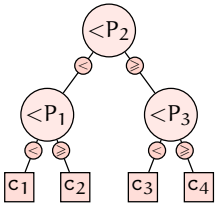
- $2 n \ln(n)$ comparisons (without sampling)

Fair Competition?

Consider the #comparisons again.

situation for YBB similar

4-way Quicksort



- Always need 2 cmps to classify.

$\rightsquigarrow \frac{24}{13} n \ln(n)$ cmps (without sampling)

≈ 1.846

Classic Quicksort

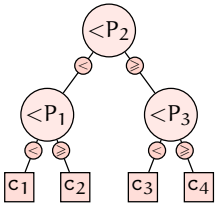
- $2 n \ln(n)$ comparisons (without sampling)

Fair Competition?

Consider the #comparisons again.

situation for YBB similar

4-way Quicksort



- Always need 2 cmps to classify.

$\rightsquigarrow \frac{24}{13} n \ln(n)$ cmps (without sampling)

≈ 1.846

VS.

Classic Quicksort

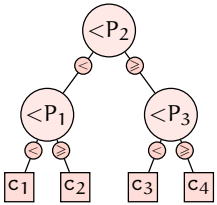
- $2 n \ln(n)$ comparisons (without sampling)

Fair Competition?

Consider the #comparisons again.

situation for YBB similar

4-way Quicksort



- Always need 2 cmps to classify.

$\rightsquigarrow \frac{24}{13} n \ln(n)$ cmps (without sampling)

≈ 1.846

VS.

Classic Quicksort

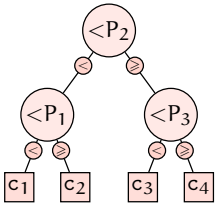
- $2 n \ln(n)$ comparisons (without sampling)

Fair Competition?

Consider the #comparisons again.

situation for YBB similar

4-way Quicksort



- Always need 2 cmps to classify.

$\rightsquigarrow \frac{24}{13} n \ln(n)$ cmps (without sampling)

≈ 1.846

VS.

Classic Quicksort

- $2 n \ln(n)$ comparisons (without sampling)

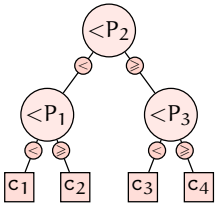
Aha! Multiway Quicksort is better!

Fair Competition?

Consider the #comparisons again.

situation for YBB similar

4-way Quicksort



- Always need 2 cmps to classify.

$\rightsquigarrow \frac{24}{13} n \ln(n)$ cmps (without sampling)

≈ 1.846

VS.

Classic Quicksort

- $2 n \ln(n)$ comparisons (without sampling)
- $\frac{12}{7} n \ln(n)$ comparisons (median-of-3)

≈ 1.714

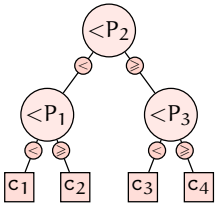
Aha! Multiway Quicksort is better!

Fair Competition?

Consider the #comparisons again.

situation for YBB similar

4-way Quicksort



- Always need 2 cmps to classify.

$\rightsquigarrow \frac{24}{13} n \ln(n)$ cmps (without sampling)

≈ 1.846

VS.

Classic Quicksort

- $2 n \ln(n)$ comparisons (without sampling)
- $\frac{12}{7} n \ln(n)$ comparisons (median-of-3)

≈ 1.714

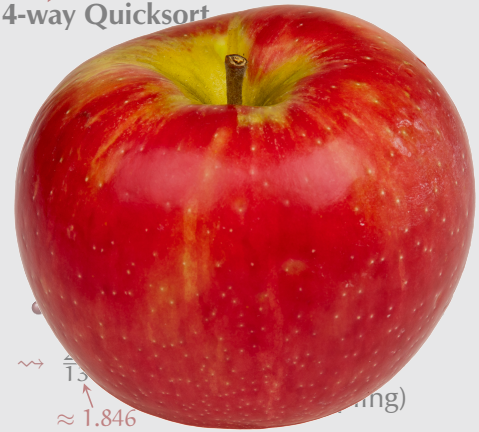
Aha! Multiway Quicksort is better!
…or not; it depends on **sampling**.

Fair Competition?

Consider the #comparisons again.

situation for YBB similar

4-way Quicksort



$\rightsquigarrow \frac{2}{13}$ ing)

≈ 1.846

VS.

Classic Quicksort

- 2



Aha! Multiwa

. . . or not; it depe

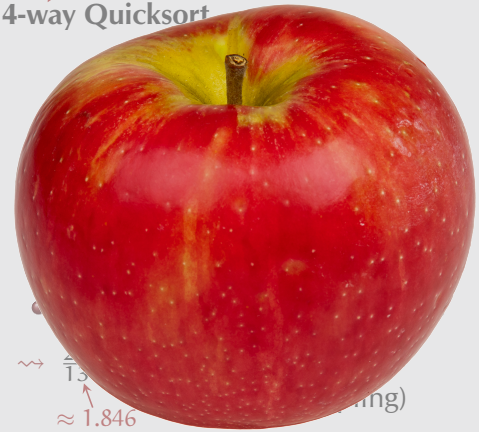
Fair Competition?

How to **separate** influence of multiway partitioning from that of sampling?

Conside

situation for YBB similar

4-way Quicksort



$\rightsquigarrow \frac{2}{13}$ ing)

≈ 1.846

VS.

Classic Quicksort

- 2



Aha! Multiwa

. . . or not; it depe

Entropy-Equivalent Sampling

Key Idea: Simulate by binary partitioning.

One round of 4-way partitioning ...

(without sampling)

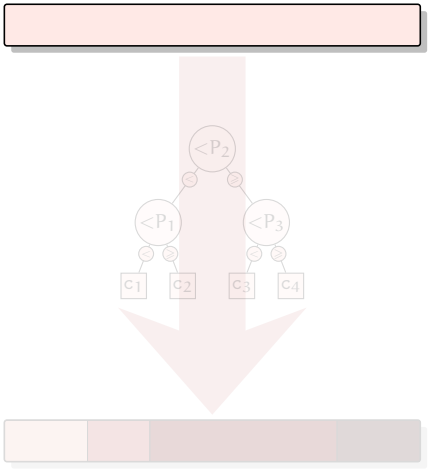
... corresponds to three binary partitionings.

Entropy-Equivalent Sampling

Key Idea: Simulate by binary partitioning.

One round of 4-way partitioning ...

(without sampling)



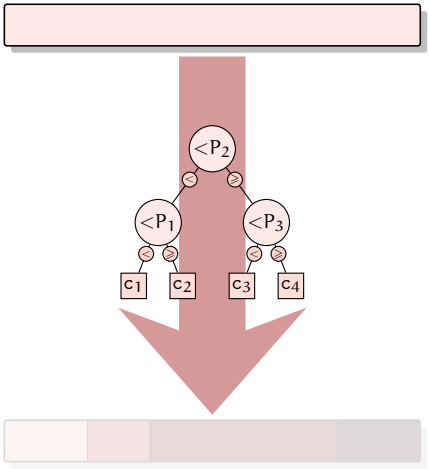
... corresponds to three binary partitionings.

Entropy-Equivalent Sampling

Key Idea: Simulate by binary partitioning.

One round of 4-way partitioning ...

(without sampling)



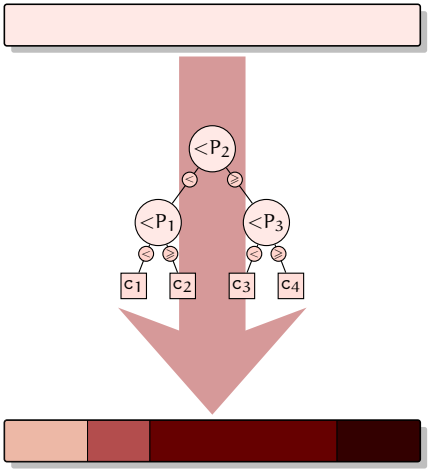
... corresponds to three binary partitionings.

Entropy-Equivalent Sampling

Key Idea: Simulate by binary partitioning.

One round of 4-way partitioning . . .

(without sampling)



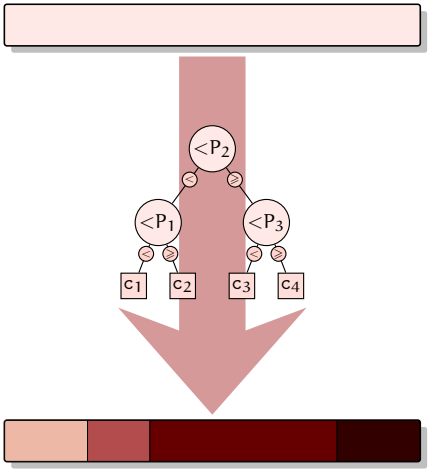
. . . corresponds to three binary partitionings.

Entropy-Equivalent Sampling

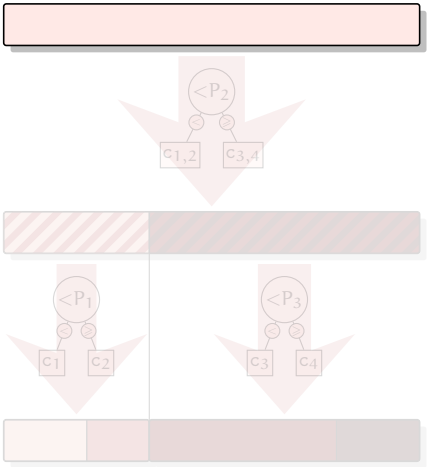
Key Idea: Simulate by binary partitioning.

One round of 4-way partitioning . . .

(without sampling)



. . . corresponds to three binary partitionings.

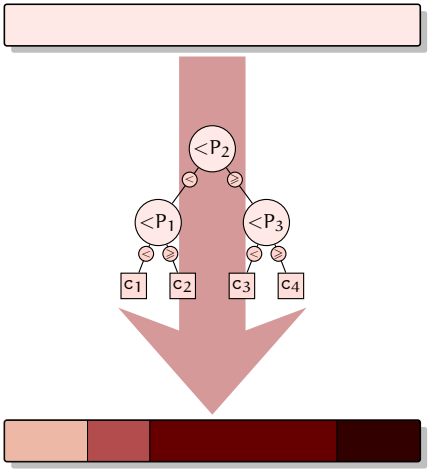


Entropy-Equivalent Sampling

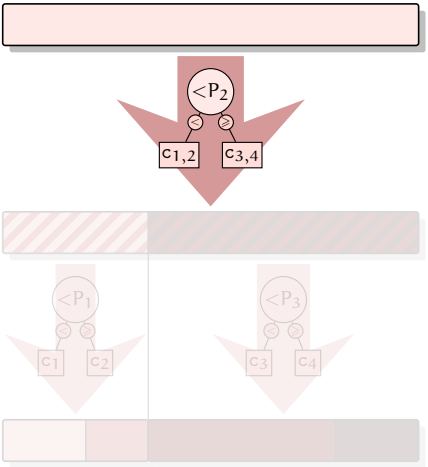
Key Idea: Simulate by binary partitioning.

One round of 4-way partitioning . . .

(without sampling)



. . . corresponds to three binary partitionings.

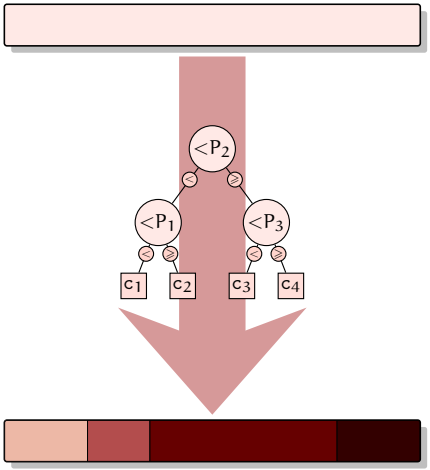


Entropy-Equivalent Sampling

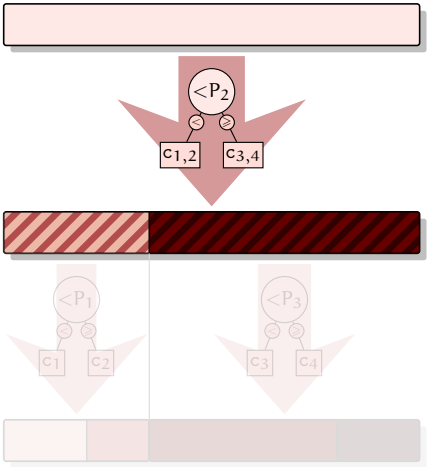
Key Idea: Simulate by binary partitioning.

One round of 4-way partitioning . . .

(without sampling)



. . . corresponds to three binary partitionings.

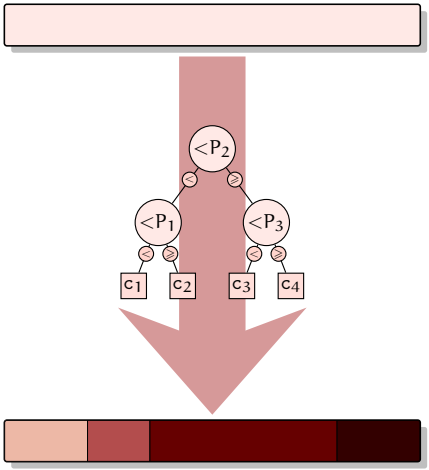


Entropy-Equivalent Sampling

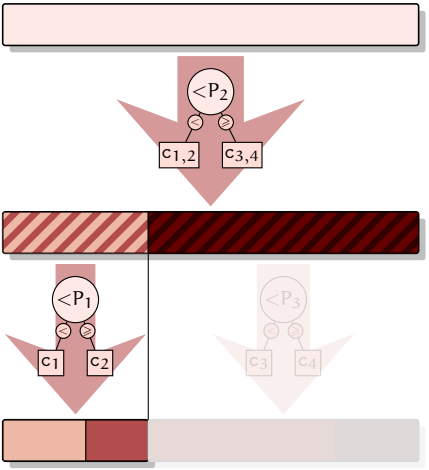
Key Idea: Simulate by binary partitioning.

One round of 4-way partitioning . . .

(without sampling)



. . . corresponds to three binary partitionings.

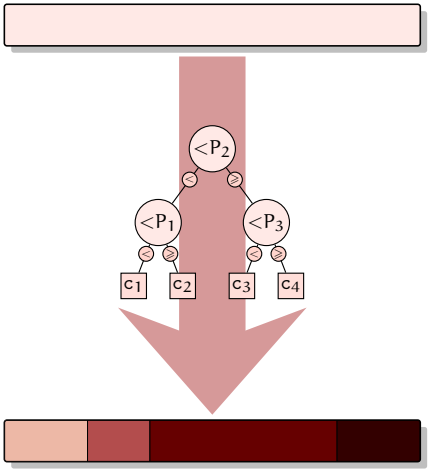


Entropy-Equivalent Sampling

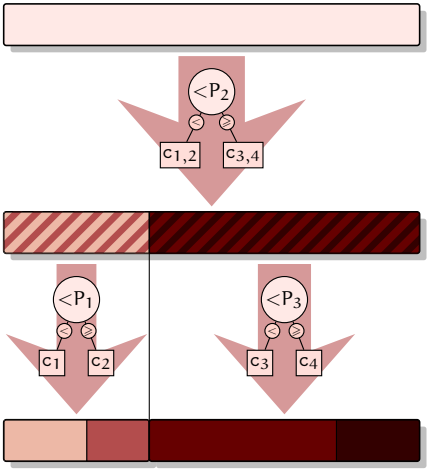
Key Idea: Simulate by binary partitioning.

One round of 4-way partitioning . . .

(without sampling)



. . . corresponds to three binary partitionings.

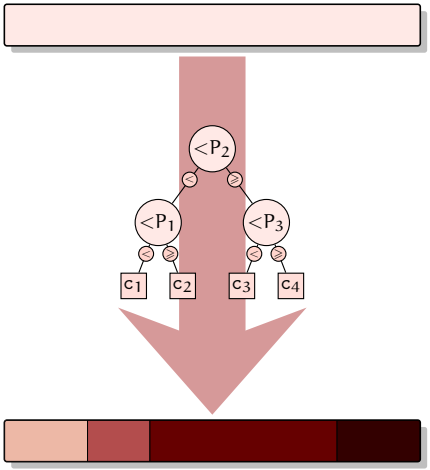


Entropy-Equivalent Sampling

Key Idea: Simulate by binary partitioning.

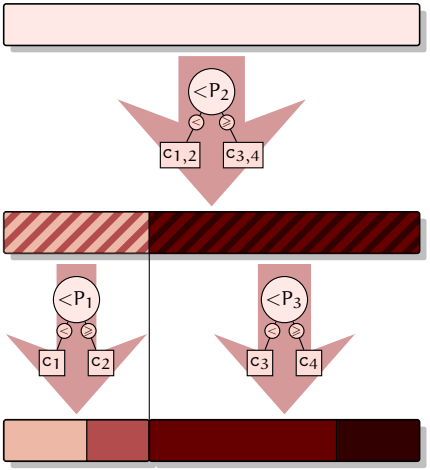
One round of 4-way partitioning ...

(without sampling)



... corresponds to three binary partitionings.

But how to choose pivots P_1, P_2, P_3?

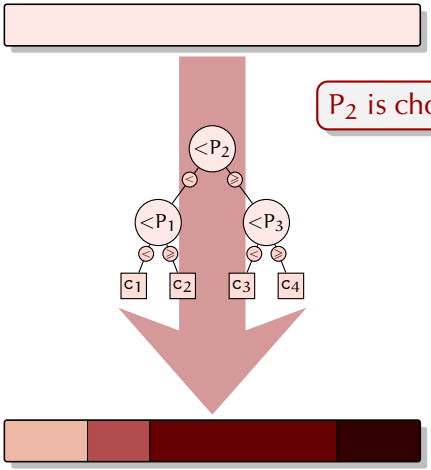


Entropy-Equivalent Sampling

Key Idea: Simulate by binary partitioning.

One round of 4-way partitioning ...

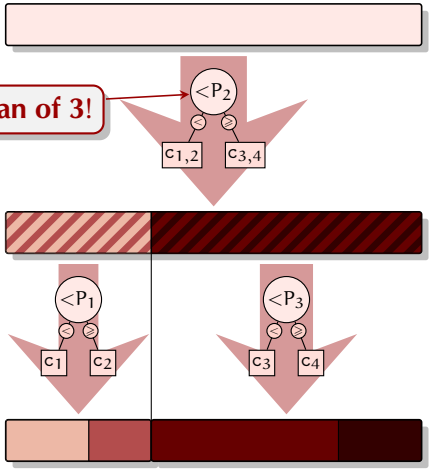
(without sampling)



... corresponds to three binary partitionings.

But how to choose pivots P_1, P_2, P_3?

P_2 is chosen as **median of 3**!



Entropy-Equivalent Sampling

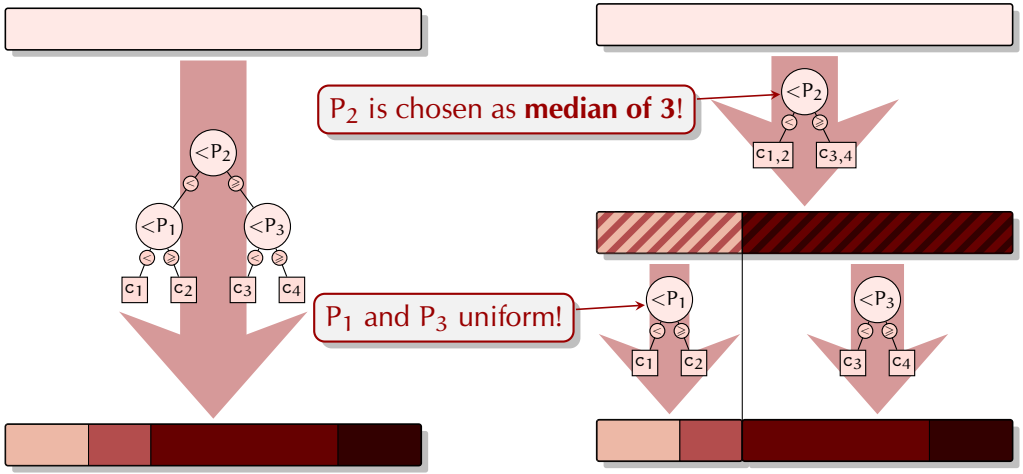
Key Idea: Simulate by binary partitioning.

One round of 4-way partitioning ...

(without sampling)

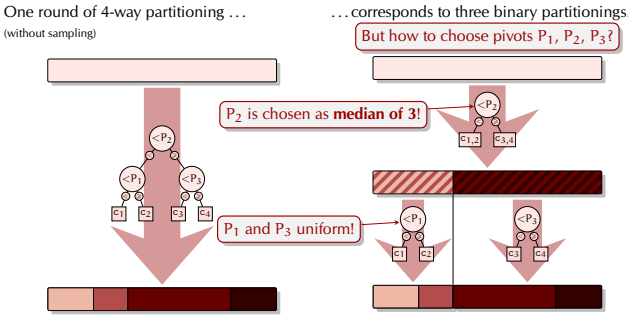
... corresponds to three binary partitionings.

But how to choose pivots P_1, P_2, P_3?



Entropy-Equivalent Sampling

Key Idea: Simulate by binary partitioning.

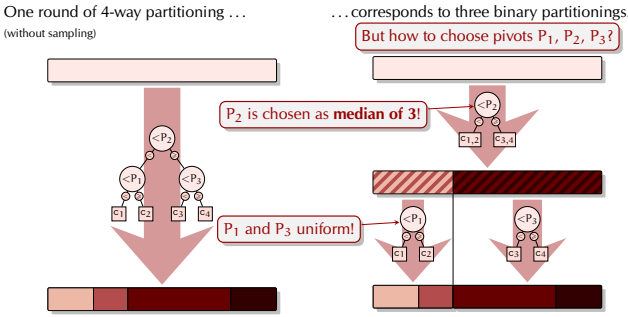


- ⇝ can **simulate** 4-way partitioning **exactly** by **alternating** between median-of-3 and no sampling.
- ⇝ **same on average**:
  1. flip a fair coin
  2. { median-of-3 for heads / no sampling for tails

random-parameter pivot sampling

Entropy-Equivalent Sampling

Key Idea: Simulate by binary partitioning.

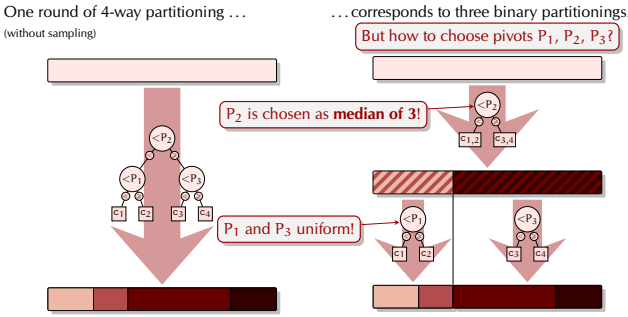


- ⇝ can **simulate** 4-way partitioning **exactly** by **alternating** between median-of-3 and no sampling.
- ⇝ **same on average**:
  1. flip a fair coin
  2. { median-of-3 for heads; no sampling for tails }

random-parameter pivot sampling

Entropy-Equivalent Sampling

Key Idea: Simulate by binary partitioning.

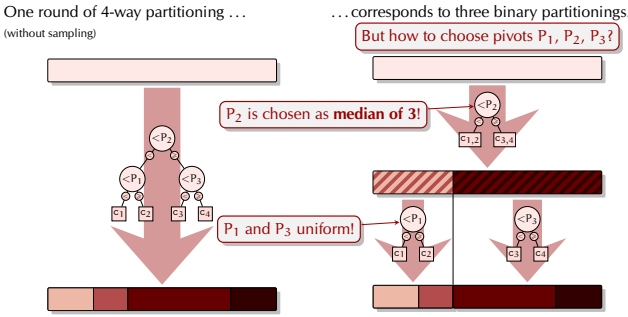


- ⇝ can **simulate** 4-way partitioning **exactly** by **alternating** between median-of-3 and no sampling.
- ⇝ **same on average**:
  1. flip a fair coin
  2. { median-of-3 for heads; no sampling for tails }

random-parameter pivot sampling

Entropy-Equivalent Sampling

Key Idea: Simulate by binary partitioning.

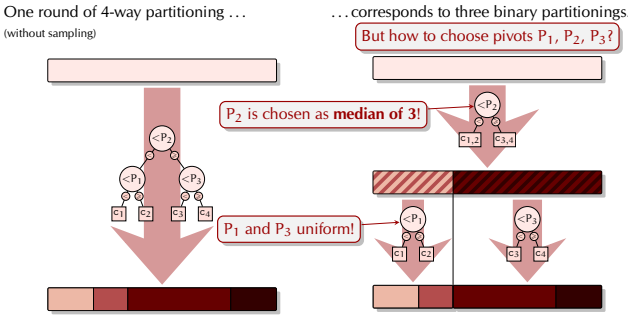


- ⇝ can **simulate** 4-way partitioning **exactly** by **alternating** between median-of-3 and no sampling.
- ⇝ **same on average**:
  1. flip a fair coin
  2. median-of-3 for heads; no sampling for tails

random-parameter pivot sampling

Entropy-Equivalent Sampling

Key Idea: Simulate by binary partitioning.

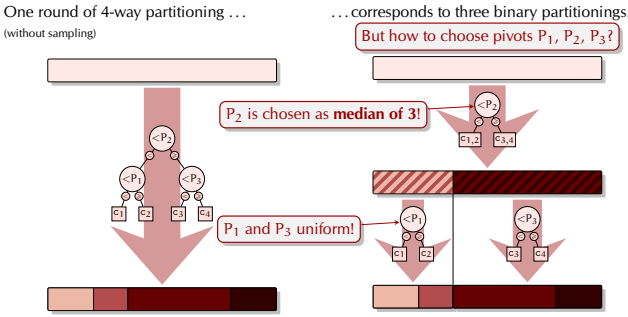


- ⇝ can **simulate** 4-way partitioning **exactly** by **alternating** between median-of-3 and no sampling.
- ⇝ **same on average**:
  1. flip a fair coin
  2. { median-of-3 for heads; no sampling for tails }

random-parameter pivot sampling

Entropy-Equivalent Sampling

Key Idea: Simulate by binary partitioning.



⇝ can **simulate** 4-way partitioning **exactly** by **alternating** between median-of-3 and no sampling.

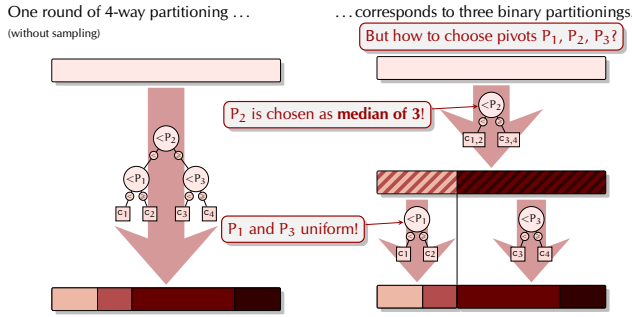
⇝ **same on average**:

1. flip a fair coin
2. median-of-3 for heads; no sampling for tails

random-parameter pivot sampling

Entropy-Equivalent Sampling

Key Idea: Simulate by binary partitioning.



- ⇝ can **simulate** 4-way partitioning **exactly** by **alternating** between median-of-3 and no sampling.
- ⇝ **same on average**:
  1. flip a fair coin
  2. { median-of-3 for heads; no sampling for tails }

random-parameter pivot sampling

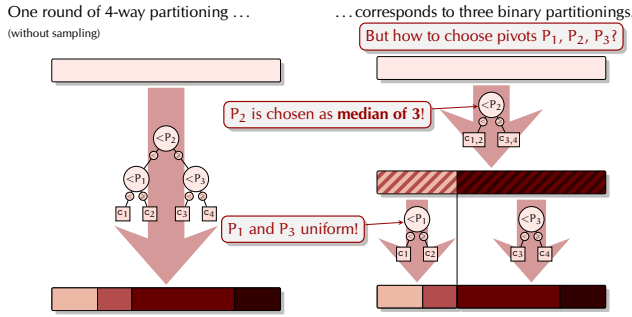
Theorem *(Entropy-Equivalent Sampling)*

For every s-way sampling parameter $\vec{t}$ there is a single-pivot random-parameter scheme with the same quality of pivots.

(formal meaning: same denominator)

Entropy-Equivalent Sampling

Key Idea: Simulate by binary partitioning.



⇝ can **simulate** 4-way partitioning **exactly** by **alternating** between median-of-3 and no sampling.

⇝ **same on average**:

1. flip a fair coin
2. $\begin{cases} \text{median-of-3} & \text{for heads} \\ \text{no sampling} & \text{for tails} \end{cases}$

random-parameter pivot sampling

Theorem ***(Entropy-Equivalent Sampling)***

For every s-way sampling parameter $\vec{t}$ there is a single-pivot random-parameter scheme with the same quality of pivots.

(formal meaning: same denominator)

Example: $\vec{t} = (1,1,1)$ with 

$$\rightsquigarrow \vec{T} = \begin{cases} (3,1) & \text{with prob. } \frac{3}{5} \\ (1,1) & \text{otherwise} \end{cases}$$

Finally: The Fair Competition

Compare multiway method to **b rounds of binary partitioning**.
b = average number of binary partitioning rounds per element.

Consider **YBB Quicksort** with $\vec{t} = (1, 1, 1)$

$\rightsquigarrow$ 2 options

1. First around P, then around Q
   $b = 1 + \mathbb{E}\left[D_1 + D_2 \right]$
2. First around Q, then around P
   $b = 1 + \mathbb{E}\left[D_1' + D_2' \right]$

$\rightsquigarrow$ $b = 1 + \frac{2}{3} = 1.\overline{6}$ (in both cases here)

- but YBB costs
  - $\frac{34}{21} \approx 1.619$ cmps/element -2.9%
  - $\frac{4}{3} = 1.\overline{3}$ scans/element -20%

Finally: The Fair Competition

Compare multiway method to **b rounds of binary partitioning**.
b = average number of binary partitioning rounds per element.

Consider **YBB Quicksort** with $\vec{t} = (1, 1, 1)$

$\rightsquigarrow$ 2 options

1. First around P, then around Q
   $b = 1 + \mathbb{E}\left[D_2 + D_3 \right]$
2. First around Q, then around P
   $b = 1 + \mathbb{E}\left[D_1 + D_2 \right]$

$\rightsquigarrow$ $b = 1 + \frac{2}{3} = 1.\overline{6}$ (in both cases here)

- but YBB costs
  - $\frac{34}{21} \approx 1.619$ cmps/element $\quad -2.9\%$
  - $\frac{4}{3} = 1.\overline{3}$ scans/element $\quad -20\%$

Finally: The Fair Competition

Compare multiway method to **b rounds of binary partitioning**.
b = average number of binary partitioning rounds per element.

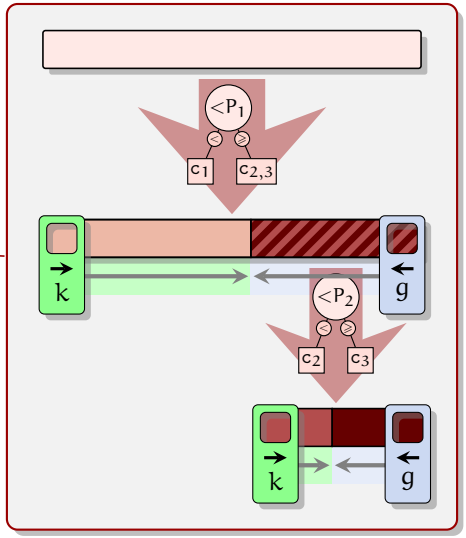
Consider **YBB Quicksort** with $\vec{t} = (1, 1, 1)$

$\rightsquigarrow$ 2 options

1. First around P, then around Q
   $b = 1 + \mathbb{E}\left[D_2 + D_3\right]$
2. First around Q, then around P
   $b = 1 + \mathbb{E}\left[D_1 + D_2\right]$

$\rightsquigarrow$ $b = 1 + \frac{2}{3} = 1.\overline{6}$ (in both cases here)

- but YBB costs
  - $\frac{34}{21} \approx 1.619$ cmps/element +2.9%
  - $\frac{4}{3} = 1.\overline{3}$ scans/element +20%



Finally: The Fair Competition

Compare multiway method to **b rounds of binary partitioning**.
b = average number of binary partitioning rounds per element.

Consider **YBB Quicksort** with $\vec{t} = (1, 1, 1)$

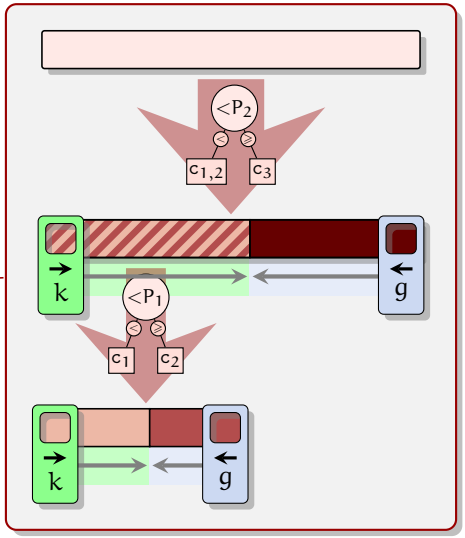
$\rightsquigarrow$ 2 options



1. First around P, then around Q
   $b = 1 + \mathbb{E}\left[D_2 + D_3\right]$
2. First around Q, then around P
   $b = 1 + \mathbb{E}\left[D_1 + D_2\right]$

$\rightsquigarrow$ $b = 1 + \frac{2}{3} = 1.\overline{6}$ (in both cases here)

- but YBB costs
  - $\frac{34}{21} \approx 1.619$ cmps/element +2.9%
  - $\frac{1}{3} = 1.\overline{3}$ scans/element +20%



Finally: The Fair Competition

Compare multiway method to **b rounds of binary partitioning**.
b = average number of binary partitioning rounds per element.

Consider **YBB Quicksort** with $\vec{t} = (1, 1, 1)$

$\rightsquigarrow$ 2 options

1. First around P, then around Q
   $b = 1 + \mathbb{E}\left[D_2 + D_3 \right]$



2. First around Q, then around P
   $b = 1 + \mathbb{E}\left[D_1 + D_2 \right]$



$\rightsquigarrow$ $b = 1 + \frac{2}{3} = 1.\overline{6}$ (in both cases here)

- but YBB costs
  - $\frac{34}{21} \approx 1.619$ cmps/element $\Rightarrow$ +2.9%
  - $\frac{1}{3} = 1.\overline{3}$ scans/element $\Rightarrow$ +20%

Finally: The Fair Competition

Compare multiway method to **b rounds of binary partitioning**.
b = average number of binary partitioning rounds per element.

Consider **YBB Quicksort** with $\vec{t} = (1, 1, 1)$

$\rightsquigarrow$ 2 options

1. First around P, then around Q
   $b = 1 + \mathbb{E}\left[D_2 + D_3 \right]$



2. First around Q, then around P
   $b = 1 + \mathbb{E}\left[D_1 + D_2 \right]$



$\rightsquigarrow$ $b = 1 + \frac{2}{3} = 1.\overline{6}$ (in both cases here)

- but YBB costs
  - $\frac{34}{21} \approx 1.619$ cmps/element — −2.9%
  - $\frac{4}{3} = 1.\overline{3}$ scans/element — −20%

Finally: The Fair Competition

Compare multiway method to **b rounds of binary partitioning**.
b = average number of binary partitioning rounds per element.

Consider **YBB Quicksort** with $\vec{t} = (1, 1, 1)$

$\rightsquigarrow$ 2 options

1. First around P, then around Q
   $b = 1 + \mathbb{E}\left[D_2 + D_3 \right]$



2. First around Q, then around P
   $b = 1 + \mathbb{E}\left[D_1 + D_2 \right]$



$\rightsquigarrow$ $b = 1 + \frac{2}{3} = 1.\overline{6}$ (in both cases here)

- but YBB costs
  - $\frac{34}{21} \approx 1.619$ cmps/element **−2.9%**
  - $\frac{4}{3} = 1.\overline{3}$ scans/element −20%

Finally: The Fair Competition

Compare multiway method to **b rounds of binary partitioning**.
b = average number of binary partitioning rounds per element.

Consider **YBB Quicksort** with $\vec{t} = (1, 1, 1)$

$\rightsquigarrow$ 2 options

1. First around P, then around Q
   $b = 1 + \mathbb{E}\left[D_2 + D_3 \right]$



2. First around Q, then around P
   $b = 1 + \mathbb{E}\left[D_1 + D_2 \right]$



$\rightsquigarrow$ $b = 1 + \frac{2}{3} = 1.\overline{6}$ (in both cases here)

- but YBB costs
  - $\frac{34}{21} \approx 1.619$ cmps/element **−2.9%**
  - $\frac{4}{3} = 1.\overline{3}$ scans/element **−20%**

Finally: The Fair Competition

Compare multiway method to **b rounds of binary partitioning**.
b = average number of binary partitioning rounds per element.

Consider **YBB Quicksort** with $\vec{t} = (1, 1, 1)$

⇝ 2 options

1. First around P, then around Q
   $b = 1 + \mathbb{E}\left[\boxed{D_2} + \boxed{D_3}\right]$



2. First around Q, then around P
   $b = 1 + \mathbb{E}\left[\boxed{D_1} + \boxed{D_2}\right]$



⇝ $b = 1 + \frac{2}{3} = 1.\overline{6}$ (in both cases here)

- but YBB costs
  - $\frac{34}{21} \approx 1.619$ cmps/element **−2.9%**
  - $\frac{4}{3} = 1.\overline{3}$ scans/element **−20%**

1.34… vs. 1.40…

But: scanned-element-optimal: $\vec{t} = (0, 1, 2)$

⇝ Option 2 strictly better: $b = 1.5$

- YBB costs
  - $\frac{34}{21} \approx 1.619$ cmps/element +7.9%
  - $\frac{7}{6} = 1.1\overline{6}$ scans/element −22.2%

Finally: The Fair Competition

Compare multiway method to **b rounds of binary partitioning**.
b = average number of binary partitioning rounds per element.

Consider **YBB Quicksort** with $\vec{t} = (1, 1, 1)$

$\rightsquigarrow$ 2 options

1. First around P, then around Q
   $b = 1 + \mathbb{E}\left[\boxed{D_2} + \boxed{D_3}\right]$



2. First around Q, then around P
   $b = 1 + \mathbb{E}\left[\boxed{D_1} + \boxed{D_2}\right]$



$\rightsquigarrow$ $b = 1 + \frac{2}{3} = 1.\overline{6}$ (in both cases here)

- but YBB costs
  - $\frac{34}{21} \approx 1.619$ cmps/element **−2.9%**
  - $\frac{4}{3} = 1.\overline{3}$ scans/element **−20%**

But: scanned-element-optimal: $\vec{t} = (0, 1, 2)$

1. $\rightsquigarrow b = 1.8\overline{3}$

$\rightsquigarrow$ Option 2 strictly better: $b = 1.5$

- YBB costs
  - $\frac{34}{21} \approx 1.619$ cmps/element **−2.9%**
  - $\frac{7}{6} = 1.1\overline{6}$ scans/element **−22.2%**

Finally: The Fair Competition

Compare multiway method to **b rounds of binary partitioning**.
b = average number of binary partitioning rounds per element.

Consider **YBB Quicksort** with $\vec{t} = (1, 1, 1)$

$\rightsquigarrow$ 2 options

1. First around P, then around Q
   $b = 1 + \mathbb{E}\left[\boxed{D_2} + \boxed{D_3}\right]$



2. First around Q, then around P
   $b = 1 + \mathbb{E}\left[\boxed{D_1} + \boxed{D_2}\right]$



$\rightsquigarrow$ $b = 1 + \frac{2}{3} = 1.\overline{6}$ (in both cases here)

- but YBB costs
  - $\frac{34}{21} \approx 1.619$ cmps/element **−2.9%**
  - $\frac{4}{3} = 1.\overline{3}$ scans/element **−20%**

But: scanned-element-optimal: $\vec{t} = (0, 1, 2)$

$\rightsquigarrow$ Option 2 strictly better: $b = 1.5$

- YBB costs
  - $\frac{34}{21} \approx 1.619$ cmps/element +7.9%
  - $\frac{7}{6} = 1.1\overline{6}$ scans/element −22.2%

Finally: The Fair Competition

Compare multiway method to **b rounds of binary partitioning**.
b = average number of binary partitioning rounds per element.

Consider **YBB Quicksort** with $\vec{t} = (1, 1, 1)$

$\rightsquigarrow$ 2 options

1. First around P, then around Q
   $b = 1 + \mathbb{E}\left[D_2 + D_3\right]$



2. First around Q, then around P
   $b = 1 + \mathbb{E}\left[D_1 + D_2\right]$



$\rightsquigarrow$ $b = 1 + \frac{2}{3} = 1.\overline{6}$ (in both cases here)

- but YBB costs
  - $\frac{34}{21} \approx 1.619$ cmps/element **−2.9%**
  - $\frac{4}{3} = 1.\overline{3}$ scans/element **−20%**

But: scanned-element-optimal: $\vec{t} = (0, 1, 2)$

$\rightsquigarrow$ Option 2 strictly better: $b = 1.5$

- YBB costs
  - $\frac{34}{21} \approx 1.619$ cmps/element +7.9%
  - $\frac{7}{6} = 1.1\overline{6}$ scans/element −22.2%

Finally: The Fair Competition

Compare multiway method to **b rounds of binary partitioning**.
b = average number of binary partitioning rounds per element.

Consider **YBB Quicksort** with $\vec{t} = (1, 1, 1)$

$\rightsquigarrow$ 2 options

1. First around P, then around Q
   $b = 1 + \mathbb{E}\left[D_2 + D_3 \right]$



2. First around Q, then around P
   $b = 1 + \mathbb{E}\left[D_1 + D_2 \right]$



$\rightsquigarrow$ $b = 1 + \frac{2}{3} = 1.\overline{6}$ (in both cases here)

- but YBB costs
  - $\frac{34}{21} \approx 1.619$ cmps/element -2.9%
  - $\frac{4}{3} = 1.\overline{3}$ scans/element -20%

But: scanned-element-optimal: $\vec{t} = (0, 1, 2)$

$\rightsquigarrow$ Option 2 strictly better: $b = 1.5$

- YBB costs
  - $\frac{34}{21} \approx 1.619$ cmps/element $+7.9\%$
  - $\frac{7}{6} = 1.1\overline{6}$ scans/element -22.2%

Finally: The Fair Competition

Compare multiway method to **b rounds of binary partitioning**.
b = average number of binary partitioning rounds per element

Consider **YBB Quicksort** with $\vec{t} = (1, 1, 1)$

$\rightsquigarrow$ 2 options

1. First around P, then around Q
   $b = 1 + \mathbb{E}\left[D_2 + D_3\right]$



2. First around Q, then around P
   $b = 1 + \mathbb{E}\left[D_1 + D_2\right]$



$\rightsquigarrow$ $b = 1 + \frac{2}{3} = 1.\overline{6}$ (in both cases here)

- but YBB costs
  - $\frac{34}{21} \approx 1.619$ cmps/element **−2.9%**
  - $\frac{4}{3} = 1.\overline{3}$ scans/element **−20%**

General Observation

- **Comparisons:** hardly any savings
  if k grows, often even costly
- **Scanned Elements:** significant savings
  large k helps, need asymmetric $\vec{t}$

But: scanned-element-optimal: $\vec{t} = (0, 1, 2)$

$\rightsquigarrow$ Option 2 strictly better: $b = 1.5$

- YBB costs
  - $\frac{34}{21} \approx 1.619$ cmps/element +7.9%
  - $\frac{7}{6} = 1.1\overline{6}$ scans/element **−22.2%**

Conclusion

You have seen:

1. tricky asymmetries in *YBB Quicksort*
   - ⇝ saves a few comparisons
     - **but:** savings vanish with pivot sampling
2. *scanned elements*
   - simple cost model for memory transfer
   - significant savings possible with multiway partitioning
3. *entropy-equivalent sampling*
   - separate effects: better pivots vs. partitioning method
   - ⇝ fair comparison

Conclusion

You have seen:

1. tricky asymmetries in *YBB Quicksort*
   - ⇝ saves a few comparisons
     - **but:** savings vanish with pivot sampling
2. *scanned elements*
   - simple cost model for memory transfer
   - significant savings possible with multiway partitioning
3. *entropy-equivalent sampling*
   - separate effects: better pivots vs. partitioning method
   - ⇝ fair comparison

Conclusion

You have seen:

1. tricky asymmetries in *YBB Quicksort*
   - ⇝ saves a few comparisons
     - **but:** savings vanish with pivot sampling
2. *scanned elements*
   - simple cost model for memory transfer
   - significant savings possible with multiway partitioning
3. *entropy-equivalent sampling*
   - separate effects: better pivots vs. partitioning method
   - ⇝ fair comparison

Conclusion

You have seen:

1. tricky asymmetries in *YBB Quicksort*
   - ⇝ saves a few comparisons
     - **but:** savings vanish with pivot sampling
2. *scanned elements*
   - simple cost model for memory transfer
   - significant savings possible with multiway partitioning
3. *entropy-equivalent sampling*
   - separate effects: better pivots vs. partitioning method
   - ⇝ fair comparison

Conclusion

You have seen:

1. tricky asymmetries in *YBB Quicksort*
   - ⇝ saves a few comparisons
     - **but:** savings vanish with pivot sampling
2. *scanned elements*
   - simple cost model for memory transfer
   - significant savings possible with multiway partitioning
3. *entropy-equivalent sampling*
   - separate effects: better pivots vs. partitioning method
   - ⇝ fair comparison

Conclusion

You have seen:

1. tricky asymmetries in *YBB Quicksort*
   - ⇝ saves a few comparisons
     - **but:** savings vanish with pivot sampling
2. *scanned elements*
   - simple cost model for memory transfer
   - significant savings possible with multiway partitioning
3. *entropy-equivalent sampling*
   - separate effects: better pivots vs. partitioning method
   - ⇝ fair comparison

Conclusion

You have seen:

1. tricky asymmetries in *YBB Quicksort*
   - ⇝ saves a few comparisons
   - **but:** savings vanish with pivot sampling
2. *scanned elements*
   - simple cost model for memory transfer
   - significant savings possible with multiway partitioning ← genuine potential of multiway partitioning!
3. *entropy-equivalent sampling*
   - separate effects: better pivots vs. partitioning method
   - ⇝ fair comparison

Conclusion

You have seen:

1. tricky asymmetries in *YBB Quicksort*
   - ⇝ saves a few comparisons
   - **but:** savings vanish with pivot sampling
2. *scanned elements*
   - simple cost model for memory transfer
   - significant savings possible with multiway partitioning ← genuine potential of multiway partitioning!
3. *entropy-equivalent sampling*
   - separate effects: better pivots vs. partitioning method
   - ⇝ fair comparison

Conclusion

You have seen:

1. tricky asymmetries in *YBB Quicksort*
   - ⇝ saves a few comparisons
     - **but:** savings vanish with pivot sampling
2. *scanned elements*
   - simple cost model for memory transfer
   - significant savings possible with multiway partitioning ← genuine potential of multiway partitioning!
3. *entropy-equivalent sampling*
   - separate effects: better pivots vs. partitioning method
   - ⇝ fair comparison

Conclusion

You have seen:

1. tricky asymmetries in *YBB Quicksort*
   - ⇝ saves a few comparisons
     - **but:** savings vanish with pivot sampling
2. *scanned elements*
   - simple cost model for memory transfer
   - significant savings possible with multiway partitioning ← genuine potential of multiway partitioning!
3. *entropy-equivalent sampling*
   - separate effects: better pivots vs. partitioning method
   - ⇝ fair comparison

Conclusion

You have seen:

1. tricky asymmetries in *YBB Quicksort*
   - ⇝ saves a few comparisons
   - **but:** savings vanish with pivot sampling
2. *scanned elements*
   - simple cost model for memory transfer
   - significant savings possible with multiway partitioning ← genuine potential of multiway partitioning!
3. *entropy-equivalent sampling*
   - separate effects: better pivots vs. partitioning method
   - ⇝ fair comparison

You have *not* seen:

- Branch mispredictions

- Treatment of equal keys

- Intriguing interplay of parameters

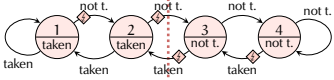
Conclusion

You have seen:

1. tricky asymmetries in *YBB Quicksort*
   - ⇝ saves a few comparisons
   - **but:** savings vanish with pivot sampling
2. *scanned elements*
   - simple cost model for memory transfer
   - significant savings possible with multiway partitioning ← genuine potential of multiway partitioning!
3. *entropy-equivalent sampling*
   - separate effects: better pivots vs. partitioning method
   - ⇝ fair comparison

You have *not* seen:

- Branch mispredictions



- Treatment of equal keys

- Intriguing interplay of parameters

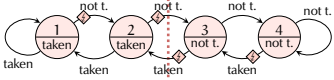
Conclusion

You have seen:

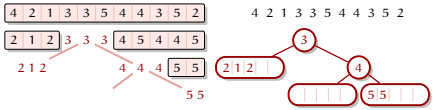
1. tricky asymmetries in *YBB Quicksort*
   - ⇝ saves a few comparisons
     - **but:** savings vanish with pivot sampling
2. *scanned elements*
   - simple cost model for memory transfer
   - significant savings possible with multiway partitioning ← genuine potential of multiway partitioning!
3. *entropy-equivalent sampling*
   - separate effects: better pivots vs. partitioning method
   - ⇝ fair comparison

You have *not* seen:

- Branch mispredictions



- Treatment of equal keys



- Intriguing interplay of parameters

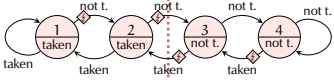
Conclusion

You have seen:

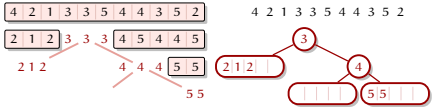
1. tricky asymmetries in *YBB Quicksort*
   - ⇝ saves a few comparisons
   - **but:** savings vanish with pivot sampling
2. *scanned elements*
   - simple cost model for memory transfer
   - significant savings possible with multiway partitioning ← genuine potential of multiway partitioning!
3. *entropy-equivalent sampling*
   - separate effects: better pivots vs. partitioning method
   - ⇝ fair comparison

You have *not* seen:

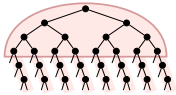
- Branch mispredictions



- Treatment of equal keys



- Intriguing interplay of parameters





Icons made by *Freepik*, *Gregor Cresnar* and *Madebyoliver* from `www.flaticon.com`.

Mousetrap Image from `bobbleheaddad.com`
Top Trumps Card from `winningmoves.co.uk`.