BlockQuicksort: Avoiding Branch Mispredictions in Quicksort

Stefan Edelkamp[1] and Armin Weiß[2,3]

[1]TZI, Universität Bremen, Germany

[2]Stevens Institute of Technology, Hoboken, NJ, USA

[3]FMI, Universität Stuttgart, Germany

Aarhus, August 24, 2016

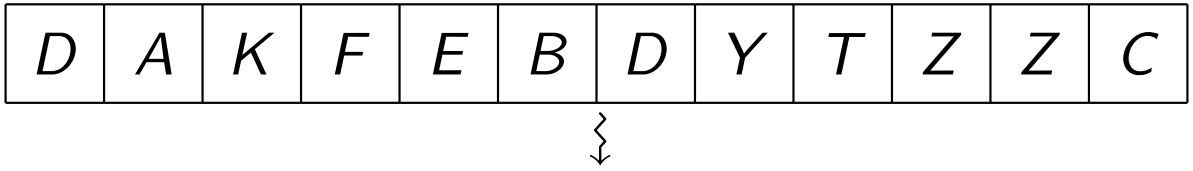
BlockQuicksort: Avoiding Branch Mispredictions in Quicksort

Outline:

- Quicksort revisited
- Effect of branch mispredictions on Quicksort
- Block partitioning
- Experimental results

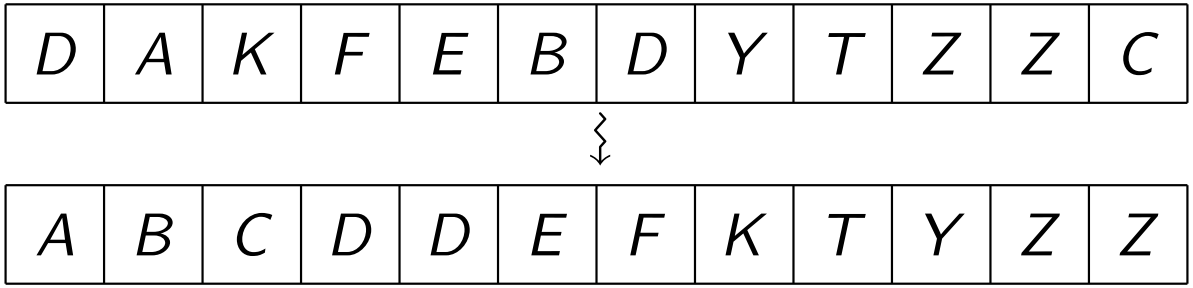
Sorting

Task: Sort a sequence of elements of some totally ordered universe.



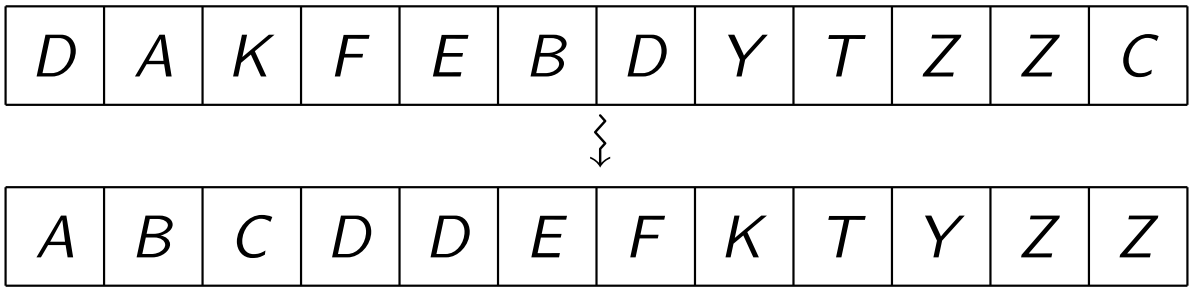
Sorting

Task: Sort a sequence of elements of some totally ordered universe.



Sorting

Task: Sort a sequence of elements of some totally ordered universe.

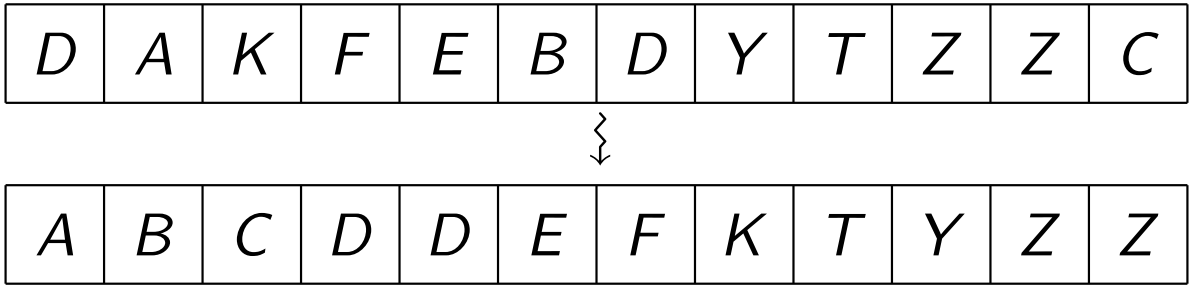


General purpuse sorting algorithm:

- for any data type
- use only pairwise comparisons
- able to handle duplicate elements

Sorting

Task: Sort a sequence of elements of some totally ordered universe.



General purpuse sorting algorithm:

- for any data type
- use only pairwise comparisons
- able to handle duplicate elements

Aim: Improve Quicksort for random inputs.

Quicksort

```
1: procedure QUICKSORT(A[ℓ, ..., r])
2:     if r > ℓ then
3:         pivot ← choosePivot(A[ℓ, ..., r])
4:         cut ← partition(A[ℓ, ..., r], pivot)
5:         Quicksort(A[ℓ, ..., cut − 1])
6:         Quicksort(A[cut, ..., r])
7:     end if
8: end procedure
```

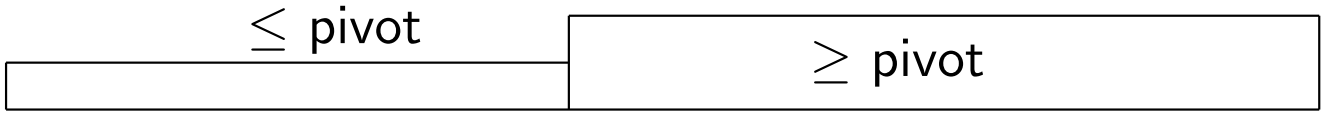
Quicksort

```
1: procedure QUICKSORT(A[ℓ, ..., r])
2:     if r > ℓ then
3:         pivot ← choosePivot(A[ℓ, ..., r])
4:         cut ← partition(A[ℓ, ..., r], pivot)
5:         Quicksort(A[ℓ, ..., cut − 1])
6:         Quicksort(A[cut, ..., r])
7:     end if
8: end procedure
```

Quicksort

```
1: procedure QUICKSORT(A[ℓ, ..., r])
2:     if r > ℓ then
3:         pivot ← choosePivot(A[ℓ, ..., r])
4:         cut ← partition(A[ℓ, ..., r], pivot)
5:         Quicksort(A[ℓ, ..., cut − 1])
6:         Quicksort(A[cut, ..., r])
7:     end if
8: end procedure
```

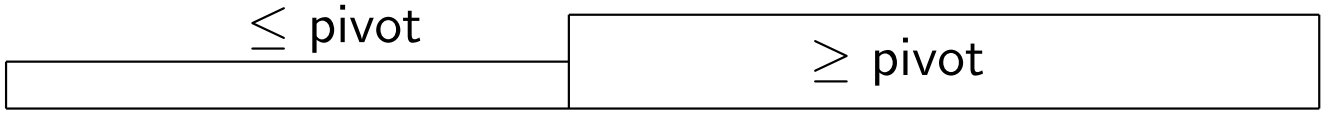
- After line 4:



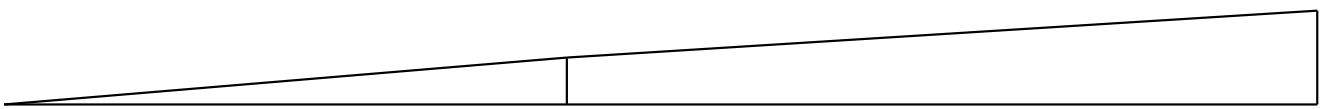
Quicksort

```
1: procedure QUICKSORT(A[ℓ, ..., r])
2:     if r > ℓ then
3:         pivot ← choosePivot(A[ℓ, ..., r])
4:         cut ← partition(A[ℓ, ..., r], pivot)
5:         Quicksort(A[ℓ, ..., cut − 1])
6:         Quicksort(A[cut, ..., r])
7:     end if
8: end procedure
```

- After line 4:



- After line 6: both parts sorted recursively with Quicksort



Properties of Quicksort

- In-place algorithm: additional space $\mathcal{O}(\log n)$ for recursion stack.

Properties of Quicksort

- In-place algorithm: additional space $\mathcal{O}(\log n)$ for recursion stack.
- $\mathcal{O}(n \log n)$ running time on average.

Properties of Quicksort

- In-place algorithm: additional space $\mathcal{O}(\log n)$ for recursion stack.
- $\mathcal{O}(n \log n)$ running time on average.

Problems of Quicksort:

- Quadratic worst case.

  Solution: Introsort (Musser 1997) – switch Heapsort in bad cases.

Properties of Quicksort

- In-place algorithm: additional space $\mathcal{O}(\log n)$ for recursion stack.
- $\mathcal{O}(n \log n)$ running time on average.

Problems of Quicksort:

- Quadratic worst case.

  Solution: Introsort (Musser 1997) – switch Heapsort in bad cases.

- Quicksort suffers from branch mispredictions in an essential way (Kaligosi, Sanders, 2006).

  Solution: Tuned Quicksort (Elmasry, Katajainen, Stenmark, 2014) – much faster than other Quicksorts, but does not allow duplicates.

Properties of Quicksort

- In-place algorithm: additional space $\mathcal{O}(\log n)$ for recursion stack.
- $\mathcal{O}(n \log n)$ running time on average.

Problems of Quicksort:

- Quadratic worst case.

  Solution: Introsort (Musser 1997) – switch Heapsort in bad cases.

- Quicksort suffers from branch mispredictions in an essential way (Kaligosi, Sanders, 2006).

  Solution: Tuned Quicksort (Elmasry, Katajainen, Stenmark, 2014) – much faster than other Quicksorts, but does not allow duplicates.

Here: general purpose algorithm with few branch mispredictions.

Pipelining and branch mispredictions

- Processor pipeline stages:
  - Instruction fetch
  - Instruction decode and register fetch
  - Actual execution
  - Memory access
  - Register write back
- 14 stages for Intel Haswell, Broadwell, Skylake processors

Pipelining and branch mispredictions

- Processor pipeline stages:
  - Instruction fetch
  - Instruction decode and register fetch
  - Actual execution
  - Memory access
  - Register write back
- 14 stages for Intel Haswell, Broadwell, Skylake processors
- Branch prediction: for every conditional statement (*if*, *while*), the processor decides in advance which branch to follow.

Pipelining and branch mispredictions

- Processor pipeline stages:
  - Instruction fetch
  - Instruction decode and register fetch
  - Actual execution
  - Memory access
  - Register write back
- 14 stages for Intel Haswell, Broadwell, Skylake processors
- Branch prediction: for every conditional statement (*if*, *while*), the processor decides in advance which branch to follow.
- Easiest branch prediction scheme: static predictor, e. g.
  - for *if* statements, take always the *if* branch,
  - for loops (*while*, *for*), assume that the loop is repeated.

Pipelining and branch mispredictions

- Processor pipeline stages:
  - Instruction fetch
  - Instruction decode and register fetch
  - Actual execution
  - Memory access
  - Register write back
- 14 stages for Intel Haswell, Broadwell, Skylake processors
- Branch prediction: for every conditional statement (*if*, *while*), the processor decides in advance which branch to follow.
- Easiest branch prediction scheme: static predictor, e. g.
  - for *if* statements, take always the *if* branch,
  - for loops (*while*, *for*), assume that the loop is repeated.
- If the execution follows the wrong branch, the content of the pipeline has to be discarded and the pipeline filled again.
  $\rightsquigarrow$ many clock-cycles wasted

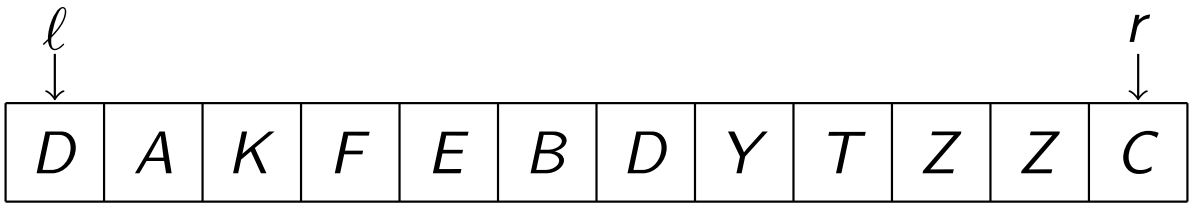
Partitioning

```
1: procedure PARTITION(A[ℓ, ..., r], pivot)
2:     while ℓ < r do
3:         while A[ℓ] < pivot do ℓ++ end while
4:         while A[r] > pivot do r−− end while
5:         if ℓ < r then swap(A[ℓ], A[r]); ℓ++; r−− end if
6:     end while
7:     return ℓ
8: end procedure
```

Partitioning

```
1: procedure PARTITION(A[ℓ, ..., r], pivot)
2:     while ℓ < r do
3:         while A[ℓ] < pivot do ℓ++ end while
4:         while A[r] > pivot do r−− end while
5:         if ℓ < r then swap(A[ℓ], A[r]); ℓ++; r−− end if
6:     end while
7:     return ℓ
8: end procedure
```

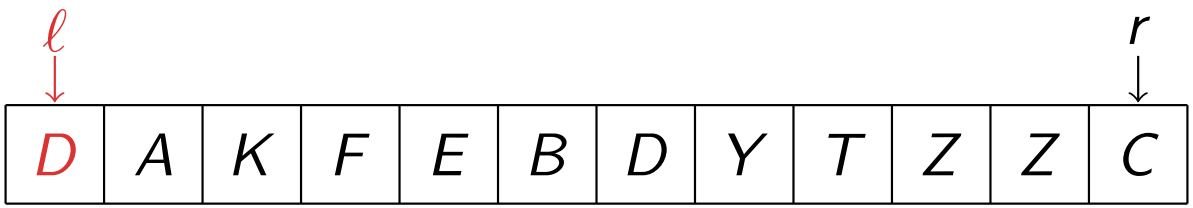
pivot = "D":



Partitioning

```
1: procedure PARTITION(A[ℓ, ..., r], pivot)
2:     while ℓ < r do
3:         while A[ℓ] < pivot do ℓ++ end while
4:         while A[r] > pivot do r−− end while
5:         if ℓ < r then swap(A[ℓ], A[r]); ℓ++; r−− end if
6:     end while
7:     return ℓ
8: end procedure
```

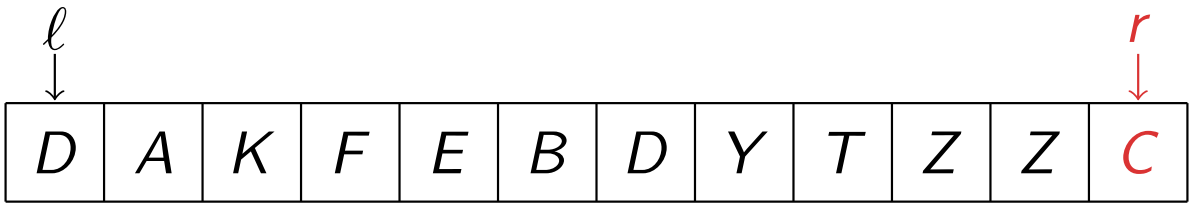
pivot = "D":



Partitioning

```
1: procedure PARTITION(A[ℓ, ..., r], pivot)
2:     while ℓ < r do
3:         while A[ℓ] < pivot do ℓ++ end while
4:         while A[r] > pivot do r−− end while
5:         if ℓ < r then swap(A[ℓ], A[r]); ℓ++; r−− end if
6:     end while
7:     return ℓ
8: end procedure
```

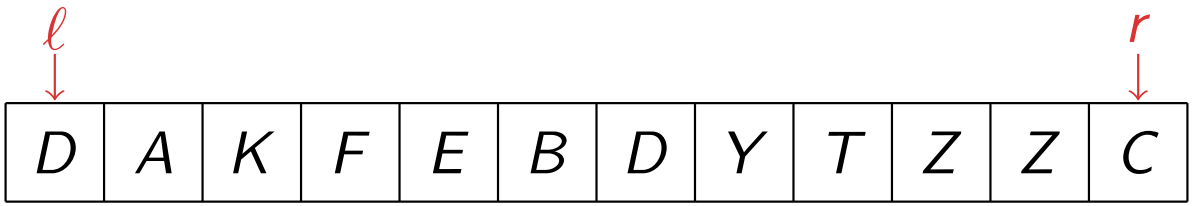
pivot = "D":



Partitioning

```
1: procedure PARTITION(A[ℓ, ..., r], pivot)
2:     while ℓ < r do
3:         while A[ℓ] < pivot do ℓ++ end while
4:         while A[r] > pivot do r−− end while
5:         if ℓ < r then swap(A[ℓ], A[r]); ℓ++; r−− end if
6:     end while
7:     return ℓ
8: end procedure
```

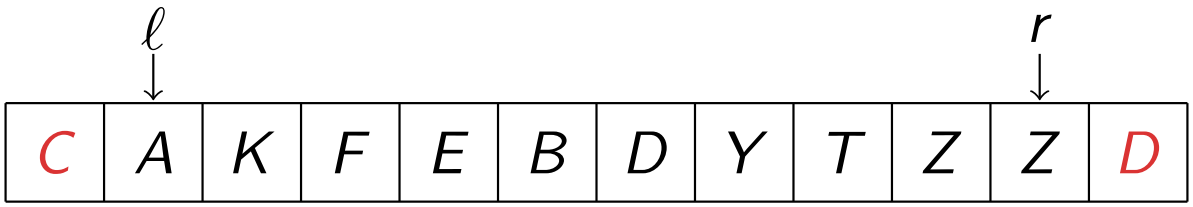
pivot = "D":



Partitioning

```
1: procedure PARTITION(A[ℓ, ..., r], pivot)
2:     while ℓ < r do
3:         while A[ℓ] < pivot do ℓ++ end while
4:         while A[r] > pivot do r−− end while
5:         if ℓ < r then swap(A[ℓ], A[r]); ℓ++; r−− end if
6:     end while
7:     return ℓ
8: end procedure
```

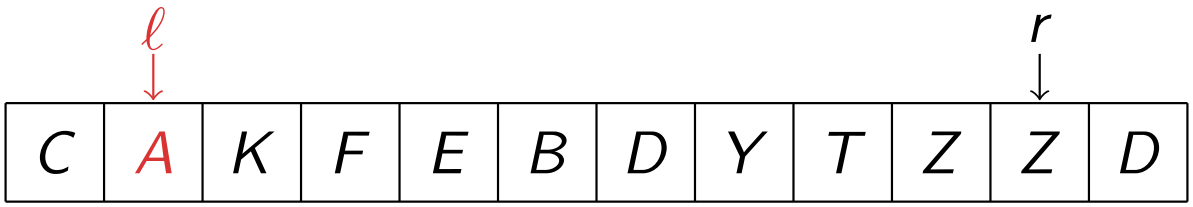
pivot = "D":



Partitioning

```
1: procedure PARTITION(A[ℓ, ..., r], pivot)
2:     while ℓ < r do
3:         while A[ℓ] < pivot do ℓ++ end while
4:         while A[r] > pivot do r−− end while
5:         if ℓ < r then swap(A[ℓ], A[r]); ℓ++; r−− end if
6:     end while
7:     return ℓ
8: end procedure
```

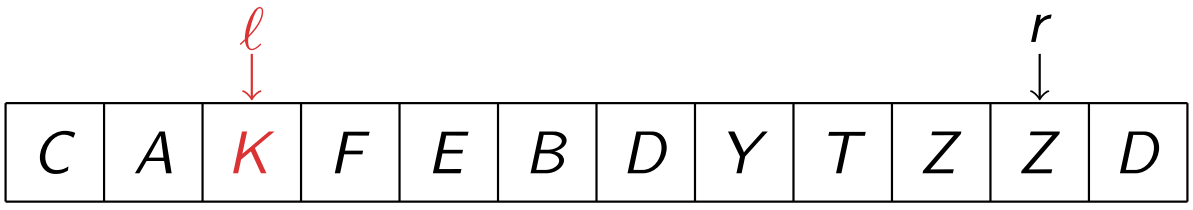
pivot = "D":



Partitioning

```
1: procedure PARTITION(A[ℓ, ..., r], pivot)
2:     while ℓ < r do
3:         while A[ℓ] < pivot do ℓ++ end while
4:         while A[r] > pivot do r−− end while
5:         if ℓ < r then swap(A[ℓ], A[r]); ℓ++; r−− end if
6:     end while
7:     return ℓ
8: end procedure
```

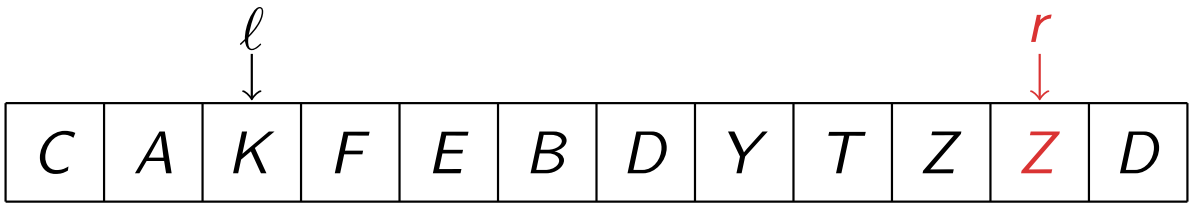
pivot = "D":



Partitioning

```
1: procedure PARTITION(A[ℓ, ..., r], pivot)
2:     while ℓ < r do
3:         while A[ℓ] < pivot do ℓ++ end while
4:         while A[r] > pivot do r−− end while
5:         if ℓ < r then swap(A[ℓ], A[r]); ℓ++; r−− end if
6:     end while
7:     return ℓ
8: end procedure
```

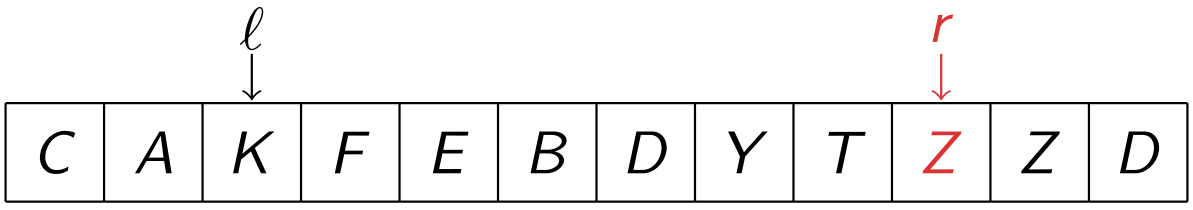
pivot = "D":



Partitioning

```
1: procedure PARTITION(A[ℓ, ..., r], pivot)
2:     while ℓ < r do
3:         while A[ℓ] < pivot do ℓ++ end while
4:         while A[r] > pivot do r−− end while
5:         if ℓ < r then swap(A[ℓ], A[r]); ℓ++; r−− end if
6:     end while
7:     return ℓ
8: end procedure
```

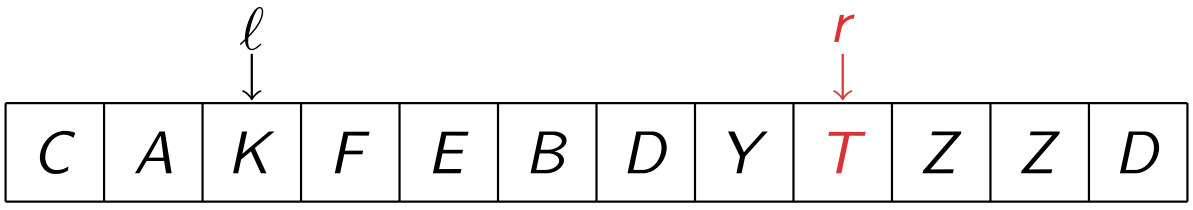
pivot = "D":



Partitioning

```
1: procedure PARTITION(A[ℓ, ..., r], pivot)
2:     while ℓ < r do
3:         while A[ℓ] < pivot do ℓ++ end while
4:         while A[r] > pivot do r−− end while
5:         if ℓ < r then swap(A[ℓ], A[r]); ℓ++; r−− end if
6:     end while
7:     return ℓ
8: end procedure
```

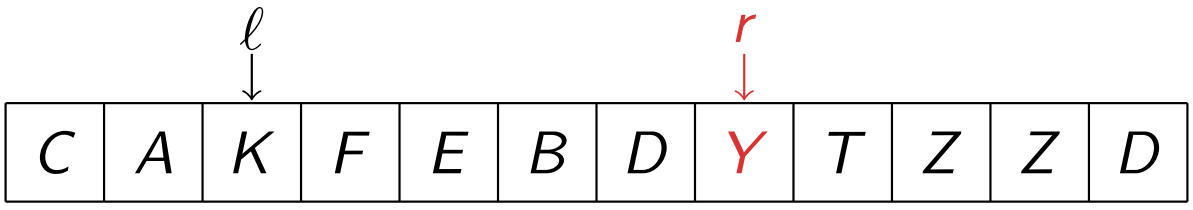
pivot = "D":



Partitioning

```
1: procedure PARTITION(A[ℓ, ..., r], pivot)
2:     while ℓ < r do
3:         while A[ℓ] < pivot do ℓ++ end while
4:         while A[r] > pivot do r−− end while
5:         if ℓ < r then swap(A[ℓ], A[r]); ℓ++; r−− end if
6:     end while
7:     return ℓ
8: end procedure
```

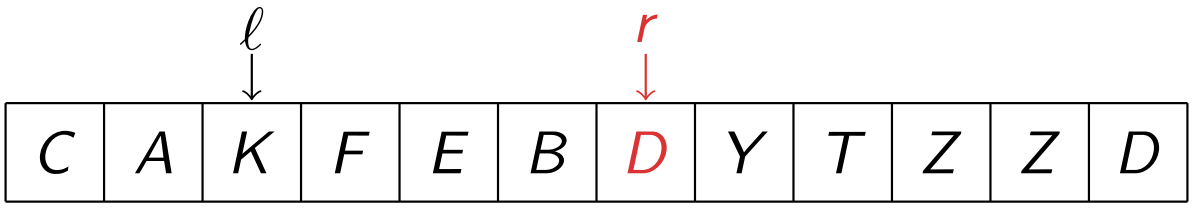
pivot = "D":



Partitioning

```
1: procedure PARTITION(A[ℓ, ..., r], pivot)
2:     while ℓ < r do
3:         while A[ℓ] < pivot do ℓ++ end while
4:         while A[r] > pivot do r−− end while
5:         if ℓ < r then swap(A[ℓ], A[r]); ℓ++; r−− end if
6:     end while
7:     return ℓ
8: end procedure
```

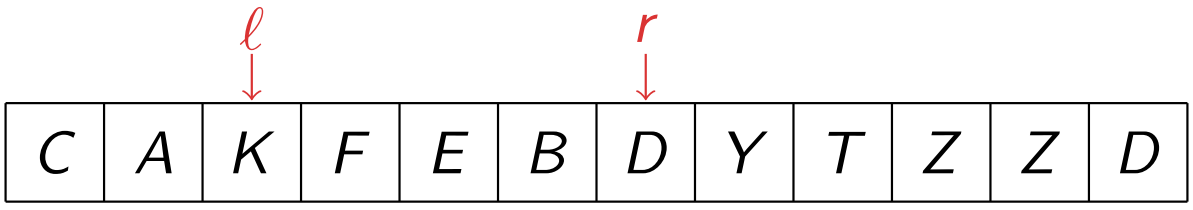
pivot = "D":



Partitioning

```
1: procedure PARTITION(A[ℓ, ..., r], pivot)
2:     while ℓ < r do
3:         while A[ℓ] < pivot do ℓ++ end while
4:         while A[r] > pivot do r−− end while
5:         if ℓ < r then swap(A[ℓ], A[r]); ℓ++; r−− end if
6:     end while
7:     return ℓ
8: end procedure
```

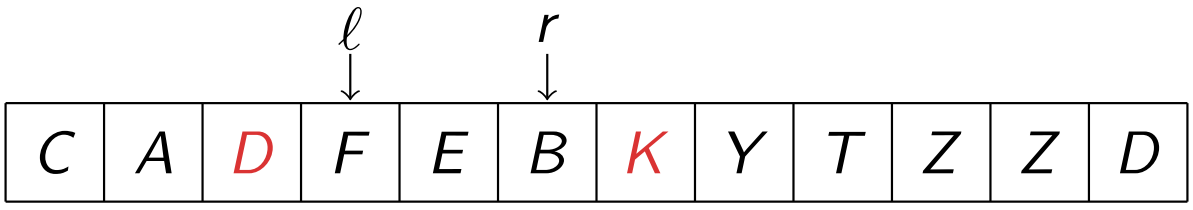
pivot = "D":



Partitioning

```
1: procedure PARTITION(A[ℓ, ..., r], pivot)
2:     while ℓ < r do
3:         while A[ℓ] < pivot do ℓ++ end while
4:         while A[r] > pivot do r−− end while
5:         if ℓ < r then swap(A[ℓ], A[r]); ℓ++; r−− end if
6:     end while
7:     return ℓ
8: end procedure
```

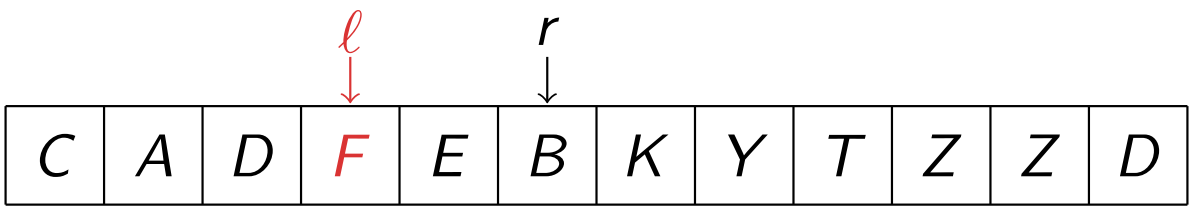
pivot = "D":



Partitioning

```
1: procedure PARTITION(A[ℓ, ..., r], pivot)
2:     while ℓ < r do
3:         while A[ℓ] < pivot do ℓ++ end while
4:         while A[r] > pivot do r−− end while
5:         if ℓ < r then swap(A[ℓ], A[r]); ℓ++; r−− end if
6:     end while
7:     return ℓ
8: end procedure
```

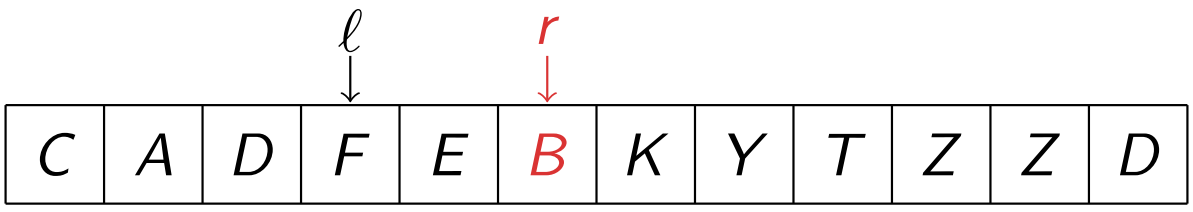
pivot = "D":



Partitioning

```
1: procedure PARTITION(A[ℓ, ..., r], pivot)
2:     while ℓ < r do
3:         while A[ℓ] < pivot do ℓ++ end while
4:         while A[r] > pivot do r−− end while
5:         if ℓ < r then swap(A[ℓ], A[r]); ℓ++; r−− end if
6:     end while
7:     return ℓ
8: end procedure
```

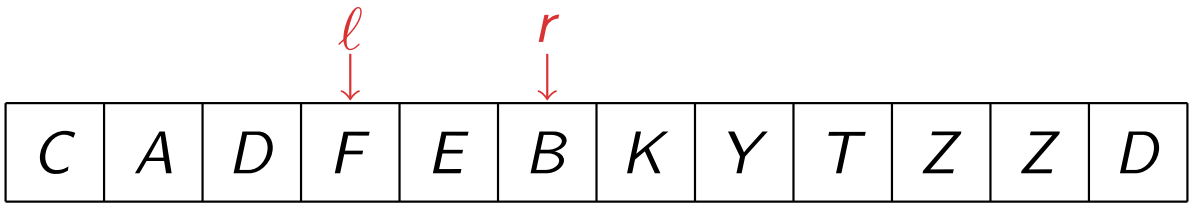
pivot = "D":



Partitioning

```
1: procedure PARTITION(A[ℓ, ..., r], pivot)
2:     while ℓ < r do
3:         while A[ℓ] < pivot do ℓ++ end while
4:         while A[r] > pivot do r−− end while
5:         if ℓ < r then swap(A[ℓ], A[r]); ℓ++; r−− end if
6:     end while
7:     return ℓ
8: end procedure
```

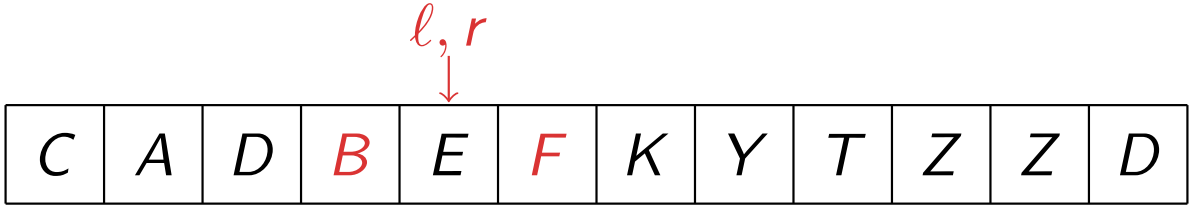
pivot = "D":



Partitioning

```
1: procedure PARTITION(A[ℓ, ..., r], pivot)
2:     while ℓ < r do
3:         while A[ℓ] < pivot do ℓ++ end while
4:         while A[r] > pivot do r−− end while
5:         if ℓ < r then swap(A[ℓ], A[r]); ℓ++; r−− end if
6:     end while
7:     return ℓ
8: end procedure
```

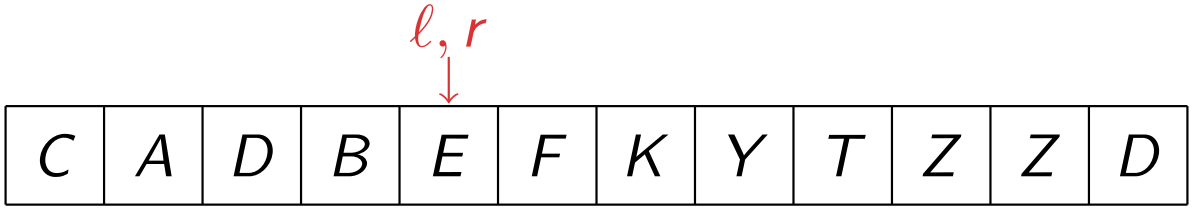
pivot = "D":



Partitioning

```
1: procedure PARTITION(A[ℓ, ..., r], pivot)
2:     while ℓ < r do
3:         while A[ℓ] < pivot do ℓ++ end while
4:         while A[r] > pivot do r−− end while
5:         if ℓ < r then swap(A[ℓ], A[r]); ℓ++; r−− end if
6:     end while
7:     return ℓ
8: end procedure
```

pivot = "D":



Partitioning is affected by branch mispredictions

```
1: procedure PARTITION(A[ℓ, ..., r], pivot)
2:     while ℓ < r do
3:         while A[ℓ] < pivot do ℓ++ end while
4:         while A[r] > pivot do r−− end while
5:         if ℓ < r then swap(A[ℓ], A[r]); ℓ++; r−− end if
6:     end while
7:     return ℓ
8: end procedure
```

- After every comparison $A[\ell] < \text{pivot}$, there are two possibilities
  - continue the loop
  - exit the loop

Partitioning is affected by branch mispredictions

```
1: procedure PARTITION(A[ℓ, ..., r], pivot)
2:     while ℓ < r do
3:         while A[ℓ] < pivot do ℓ++ end while
4:         while A[r] > pivot do r−− end while
5:         if ℓ < r then swap(A[ℓ], A[r]); ℓ++; r−− end if
6:     end while
7:     return ℓ
8: end procedure
```

- After every comparison $A[\ell] < \text{pivot}$, there are two possibilities
  - continue the loop
  - exit the loop
- if pivot $=$ median, there is a 50% chance for each
  $\rightsquigarrow$ branch predictor can only "guess"

Partitioning is affected by branch mispredictions

```
1: procedure PARTITION(A[ℓ, ..., r], pivot)
2:     while ℓ < r do
3:         while A[ℓ] < pivot do ℓ++ end while
4:         while A[r] > pivot do r−− end while
5:         if ℓ < r then swap(A[ℓ], A[r]); ℓ++; r−− end if
6:     end while
7:     return ℓ
8: end procedure
```

- After every comparison $A[\ell] < \text{pivot}$, there are two possibilities
  - continue the loop
  - exit the loop
- if pivot $=$ median, there is a 50% chance for each
  $\rightsquigarrow$ branch predictor can only "guess"
- many clockcycles wasted with refilling the pipeline.

Partitioning is affected by branch mispredictions

Theorem (Kaligosi, Sanders, 2006)

Quicksort incurs $0.35n \log n + \mathcal{O}(n)$ *branch mispredictions on average with static branch prediction.*

For comparison: Quicksort needs $1.38n \log n$ comparisons on average.

- Also, Kaligosi and Sanders established experimentally that a skewed pivot improves the running time of Quicksort.
- We repeated their experiments: best results if the $\frac{n}{6}$-th element is chosen as pivot.
  $\rightsquigarrow$ e. g. select 60 elements and choose pivot as the 10th of them.

Block Partitioning

Choose block size B (we use $B = 128$)

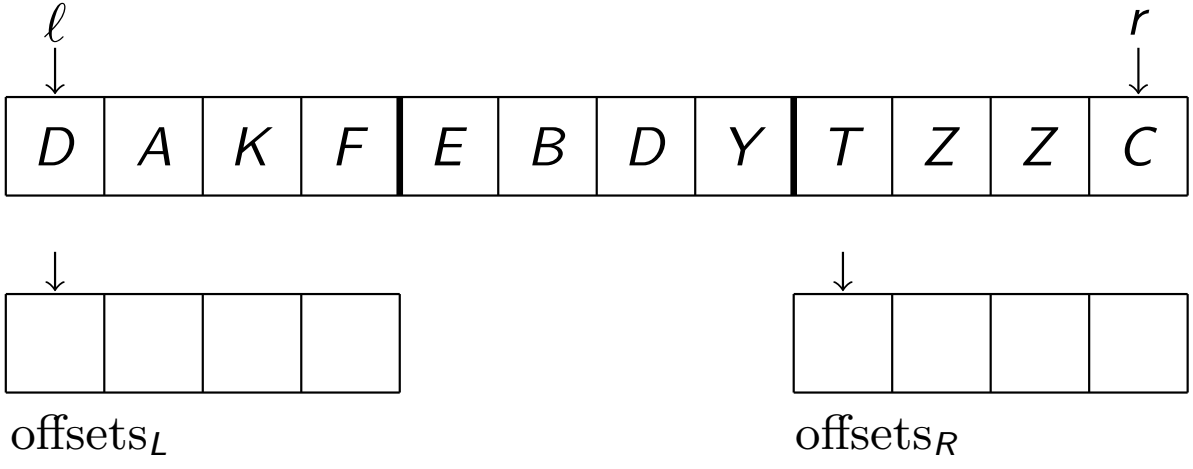
```
1:  procedure BlockPartition(A[ℓ, ..., r], pivot)
2:      integer offsets_L[0, ..., B − 1], offsets_R[0, ..., B − 1]
3:      integer start_L, start_R, num_L, num_R ← 0
4:      while r − ℓ + 1 > 2B do                          ▷ start main loop
5:          ScanLeft
6:          ScanRight
7:          Rearrange                                    ▷ swap elements
8:      end while                                        ▷ end main loop
9:      scan and rearrange remaining elements
10: end procedure
```

Block Partitioning

Choose block size B (we use $B = 128$)

```
 1: procedure BlockPartition(A[ℓ, ..., r], pivot)
 2:     integer offsets_L[0, ..., B − 1], offsets_R[0, ..., B − 1]
 3:     integer start_L, start_R, num_L, num_R ← 0
 4:     while r − ℓ + 1 > 2B do                          ▷ start main loop
 5:         ScanLeft
 6:         ScanRight
 7:         Rearrange                                    ▷ swap elements
 8:     end while                                        ▷ end main loop
 9:     scan and rearrange remaining elements
10: end procedure
```

Block size $B = 4$, pivot = "D":

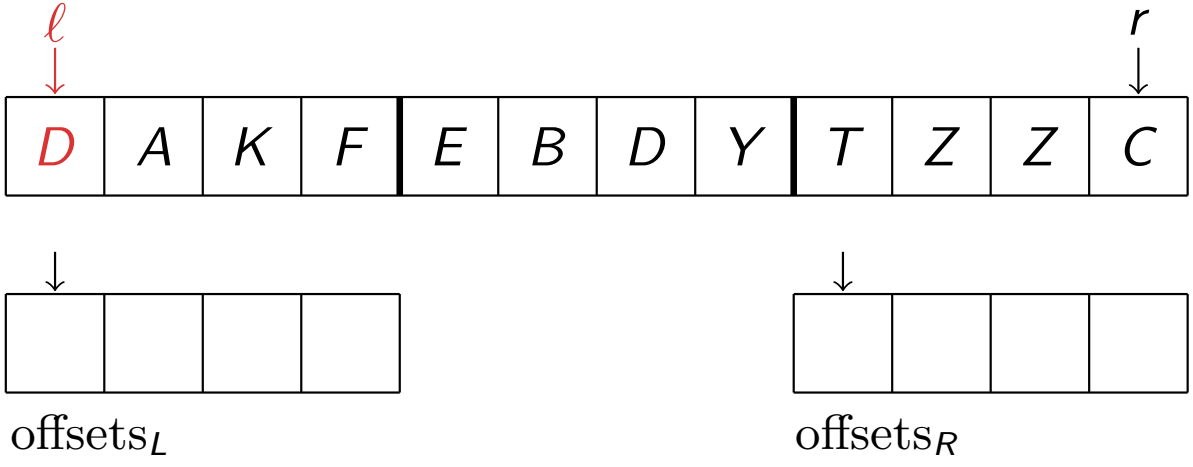


Block Partitioning

Choose block size B (we use $B = 128$)

```
 1: procedure BlockPartition(A[ℓ, ..., r], pivot)
 2:     integer offsets_L[0, ..., B − 1], offsets_R[0, ..., B − 1]
 3:     integer start_L, start_R, num_L, num_R ← 0
 4:     while r − ℓ + 1 > 2B do                          ▷ start main loop
 5:         ScanLeft
 6:         ScanRight
 7:         Rearrange                                    ▷ swap elements
 8:     end while                                        ▷ end main loop
 9:     scan and rearrange remaining elements
10: end procedure
```

Block size $B = 4$, pivot = "D":

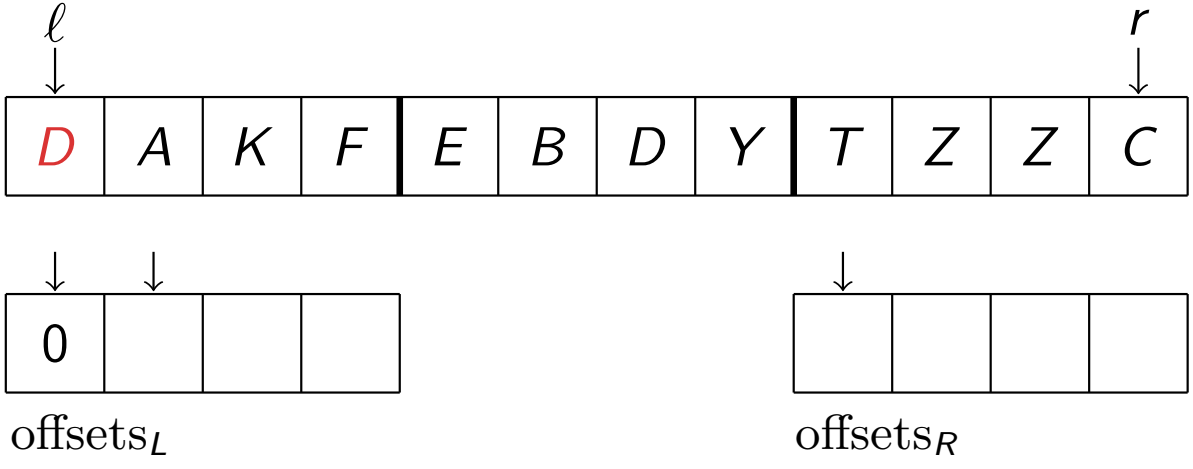


Block Partitioning

Choose block size B (we use $B = 128$)

```
 1: procedure BlockPartition(A[ℓ, ..., r], pivot)
 2:     integer offsets_L[0, ..., B − 1], offsets_R[0, ..., B − 1]
 3:     integer start_L, start_R, num_L, num_R ← 0
 4:     while r − ℓ + 1 > 2B do                          ▷ start main loop
 5:         ScanLeft
 6:         ScanRight
 7:         Rearrange                                    ▷ swap elements
 8:     end while                                        ▷ end main loop
 9:     scan and rearrange remaining elements
10: end procedure
```

Block size $B = 4$, pivot = "D":

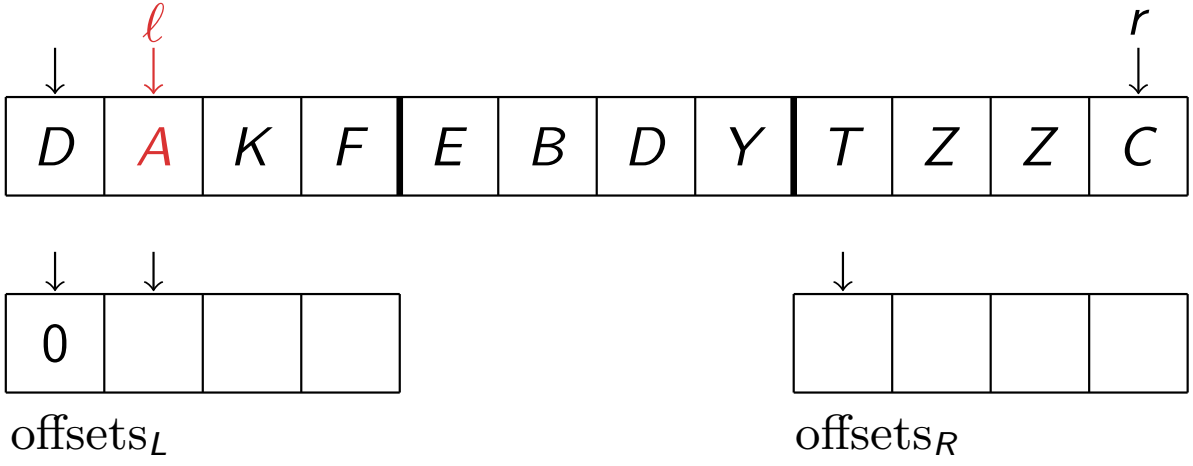


Block Partitioning

Choose block size B (we use $B = 128$)

1: **procedure** BLOCKPARTITION($A[\ell, \ldots, r]$, pivot)
2: **integer** $\text{offsets}_L[0, \ldots, B-1], \text{offsets}_R[0, \ldots, B-1]$
3: **integer** $\text{start}_L, \text{start}_R, \text{num}_L, \text{num}_R \leftarrow 0$
4: **while** $r - \ell + 1 > 2B$ **do** ▷ start main loop
5: ScanLeft
6: ScanRight
7: Rearrange ▷ swap elements
8: **end while** ▷ end main loop
9: scan and rearrange remaining elements
10: **end procedure**

Block size $B = 4$, pivot = "D":

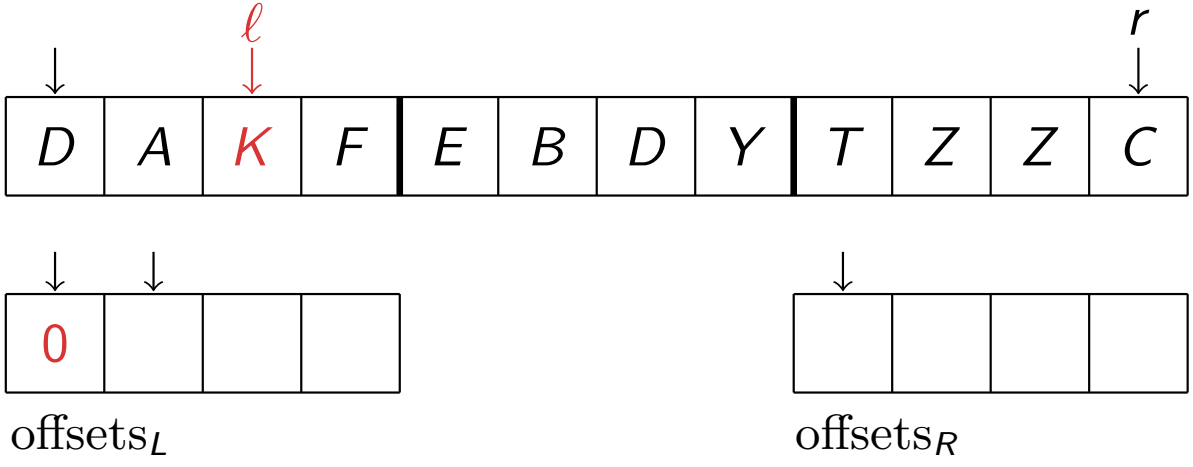


Block Partitioning

Choose block size B (we use $B = 128$)

1: **procedure** BLOCKPARTITION($A[\ell, \ldots, r]$, pivot)
2: **integer** $\text{offsets}_L[0, \ldots, B-1], \text{offsets}_R[0, \ldots, B-1]$
3: **integer** $\text{start}_L, \text{start}_R, \text{num}_L, \text{num}_R \leftarrow 0$
4: **while** $r - \ell + 1 > 2B$ **do** ▷ start main loop
5: ScanLeft
6: ScanRight
7: Rearrange ▷ swap elements
8: **end while** ▷ end main loop
9: scan and rearrange remaining elements
10: **end procedure**

Block size $B = 4$, pivot = "D":

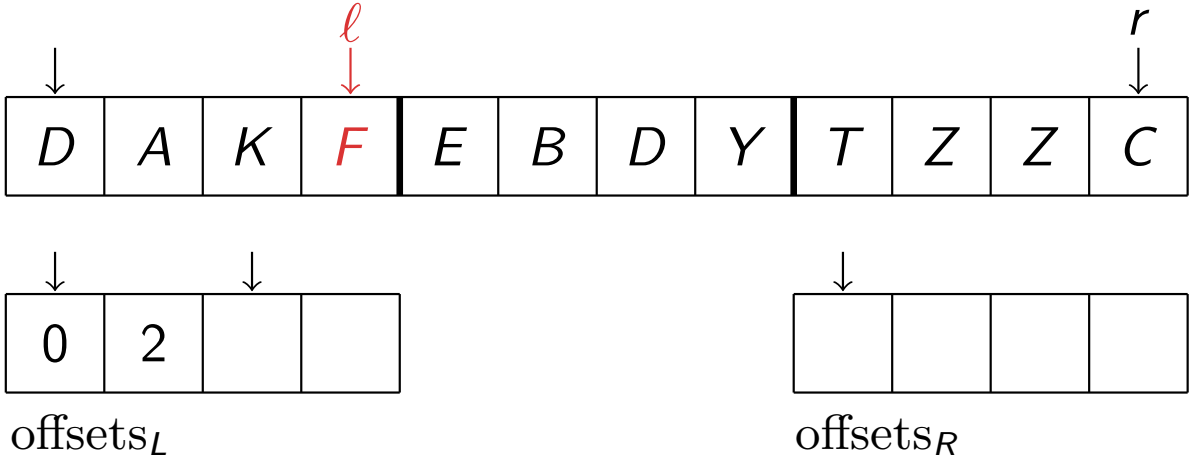


Block Partitioning

Choose block size B (we use $B = 128$)

1: **procedure** BLOCKPARTITION($A[\ell, \ldots, r]$, pivot)
2: **integer** $\text{offsets}_L[0, \ldots, B-1], \text{offsets}_R[0, \ldots, B-1]$
3: **integer** $\text{start}_L, \text{start}_R, \text{num}_L, \text{num}_R \leftarrow 0$
4: **while** $r - \ell + 1 > 2B$ **do** ▷ start main loop
5: ScanLeft
6: ScanRight
7: Rearrange ▷ swap elements
8: **end while** ▷ end main loop
9: scan and rearrange remaining elements
10: **end procedure**

Block size $B = 4$, pivot = "D":

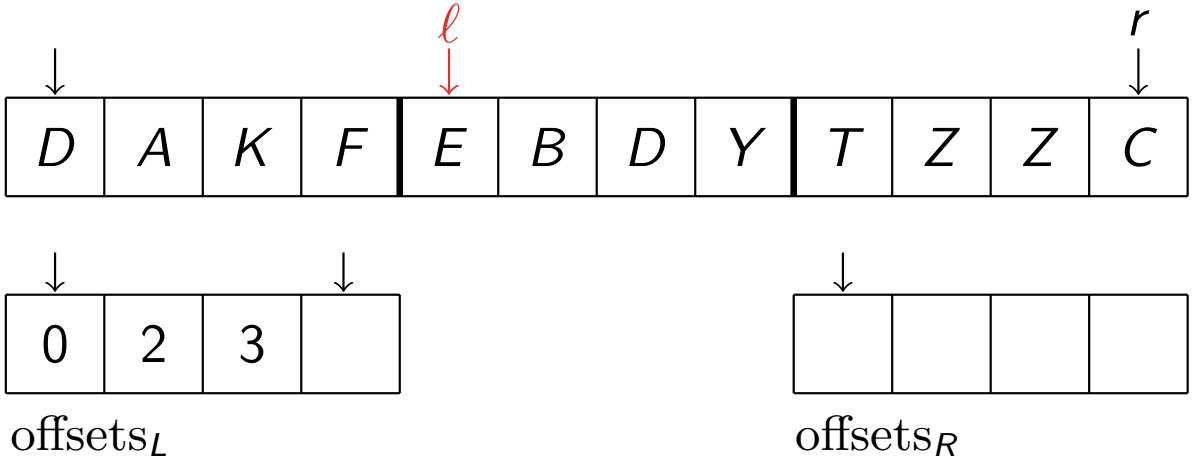


Block Partitioning

Choose block size B (we use $B = 128$)

1: **procedure** BLOCKPARTITION($A[\ell, \ldots, r]$, pivot)
2: **integer** $\text{offsets}_L[0, \ldots, B-1], \text{offsets}_R[0, \ldots, B-1]$
3: **integer** $\text{start}_L, \text{start}_R, \text{num}_L, \text{num}_R \leftarrow 0$
4: **while** $r - \ell + 1 > 2B$ **do** ▷ start main loop
5: ScanLeft
6: ScanRight
7: Rearrange ▷ swap elements
8: **end while** ▷ end main loop
9: scan and rearrange remaining elements
10: **end procedure**

Block size $B = 4$, pivot = "D":

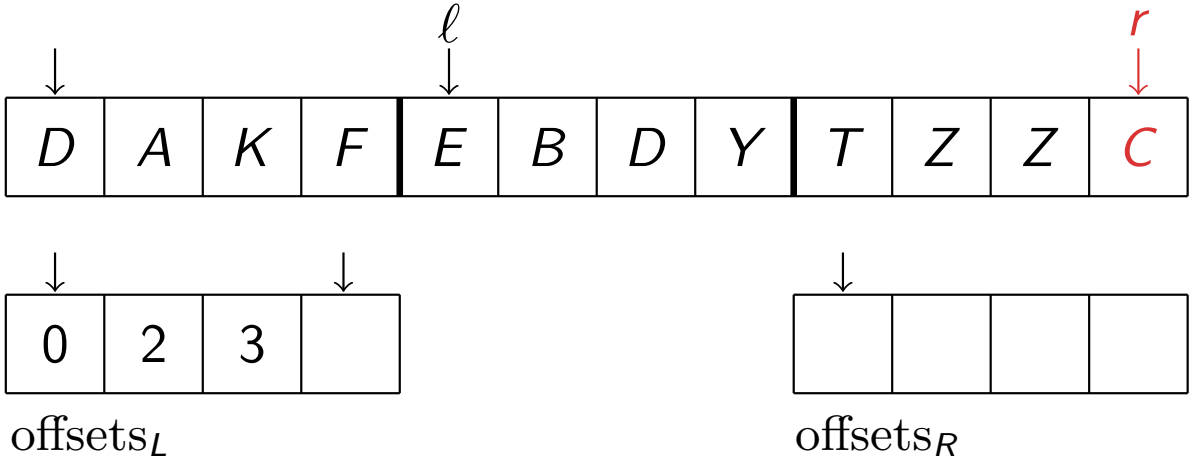


Block Partitioning

Choose block size B (we use $B = 128$)

```
 1: procedure BlockPartition(A[ℓ, ..., r], pivot)
 2:     integer offsets_L[0, ..., B − 1], offsets_R[0, ..., B − 1]
 3:     integer start_L, start_R, num_L, num_R ← 0
 4:     while r − ℓ + 1 > 2B do                          ▷ start main loop
 5:         ScanLeft
 6:         ScanRight
 7:         Rearrange                                    ▷ swap elements
 8:     end while                                        ▷ end main loop
 9:     scan and rearrange remaining elements
10: end procedure
```

Block size $B = 4$, pivot = "D":

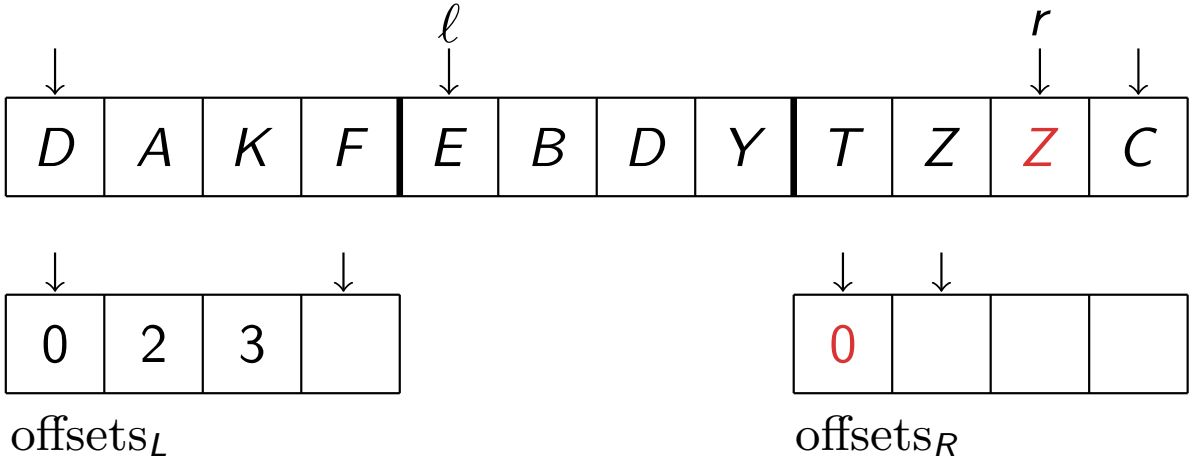


Block Partitioning

Choose block size B (we use $B = 128$)

1: **procedure** BLOCKPARTITION($A[\ell, \ldots, r]$, pivot)
2: **integer** $\text{offsets}_L[0, \ldots, B-1], \text{offsets}_R[0, \ldots, B-1]$
3: **integer** $\text{start}_L, \text{start}_R, \text{num}_L, \text{num}_R \leftarrow 0$
4: **while** $r - \ell + 1 > 2B$ **do** ▷ start main loop
5: ScanLeft
6: ScanRight
7: Rearrange ▷ swap elements
8: **end while** ▷ end main loop
9: scan and rearrange remaining elements
10: **end procedure**

Block size $B = 4$, pivot = "D":

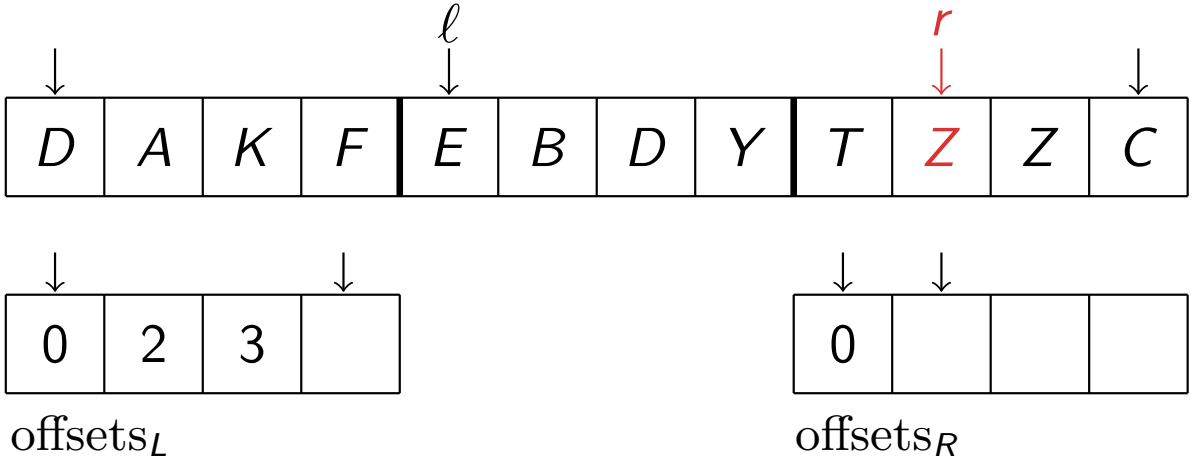


Block Partitioning

Choose block size B (we use $B = 128$)

1: **procedure** BLOCKPARTITION($A[\ell, \ldots, r]$, pivot)
2: **integer** $\text{offsets}_L[0, \ldots, B-1], \text{offsets}_R[0, \ldots, B-1]$
3: **integer** $\text{start}_L, \text{start}_R, \text{num}_L, \text{num}_R \leftarrow 0$
4: **while** $r - \ell + 1 > 2B$ **do** ▷ start main loop
5: ScanLeft
6: ScanRight
7: Rearrange ▷ swap elements
8: **end while** ▷ end main loop
9: scan and rearrange remaining elements
10: **end procedure**

Block size $B = 4$, pivot = "D":

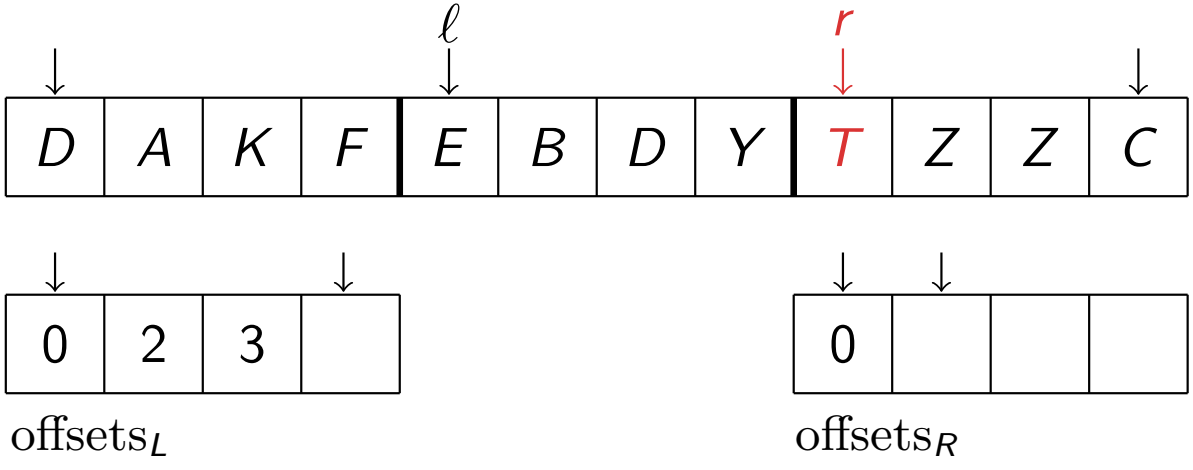


Block Partitioning

Choose block size B (we use $B = 128$)

```
 1: procedure BlockPartition(A[ℓ, ..., r], pivot)
 2:     integer offsets_L[0, ..., B − 1], offsets_R[0, ..., B − 1]
 3:     integer start_L, start_R, num_L, num_R ← 0
 4:     while r − ℓ + 1 > 2B do                          ▷ start main loop
 5:         ScanLeft
 6:         ScanRight
 7:         Rearrange                                    ▷ swap elements
 8:     end while                                        ▷ end main loop
 9:     scan and rearrange remaining elements
10: end procedure
```

Block size $B = 4$, pivot = "D":

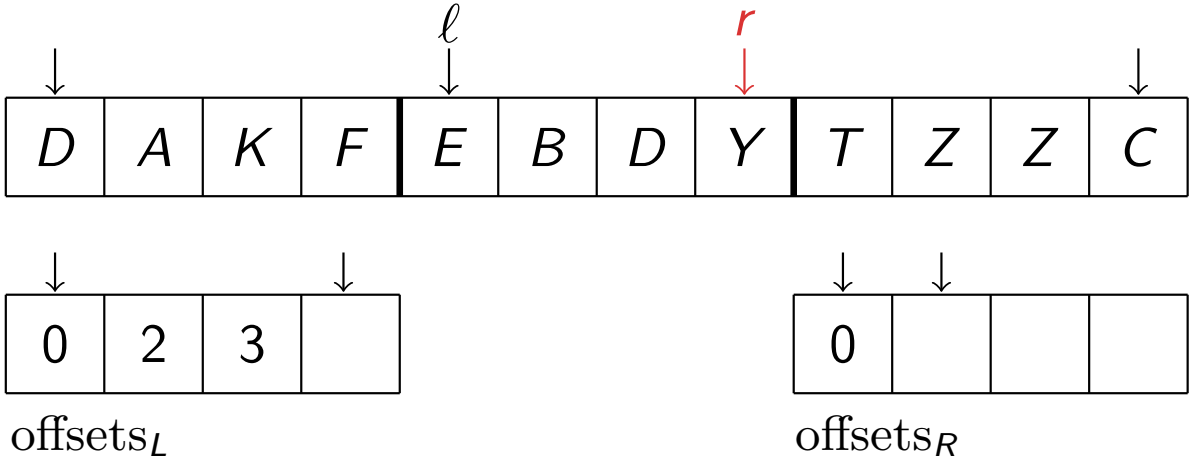


Block Partitioning

Choose block size B (we use $B = 128$)

1: **procedure** BLOCKPARTITION($A[\ell, \ldots, r]$, pivot)
2: **integer** $\text{offsets}_L[0, \ldots, B-1], \text{offsets}_R[0, \ldots, B-1]$
3: **integer** $\text{start}_L, \text{start}_R, \text{num}_L, \text{num}_R \leftarrow 0$
4: **while** $r - \ell + 1 > 2B$ **do** ▷ start main loop
5: ScanLeft
6: ScanRight
7: Rearrange ▷ swap elements
8: **end while** ▷ end main loop
9: scan and rearrange remaining elements
10: **end procedure**

Block size $B = 4$, pivot = "D":

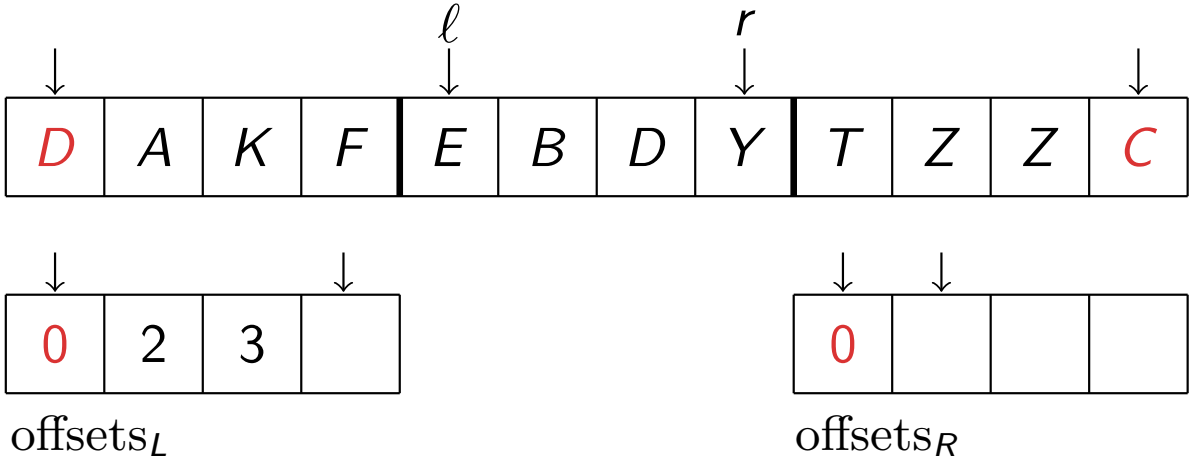


Block Partitioning

Choose block size B (we use $B = 128$)

1: **procedure** BLOCKPARTITION($A[\ell, \ldots, r]$, pivot)
2: **integer** $\text{offsets}_L[0, \ldots, B-1], \text{offsets}_R[0, \ldots, B-1]$
3: **integer** $\text{start}_L, \text{start}_R, \text{num}_L, \text{num}_R \leftarrow 0$
4: **while** $r - \ell + 1 > 2B$ **do** ▷ start main loop
5: ScanLeft
6: ScanRight
7: Rearrange ▷ swap elements
8: **end while** ▷ end main loop
9: scan and rearrange remaining elements
10: **end procedure**

Block size $B = 4$, pivot = "D":

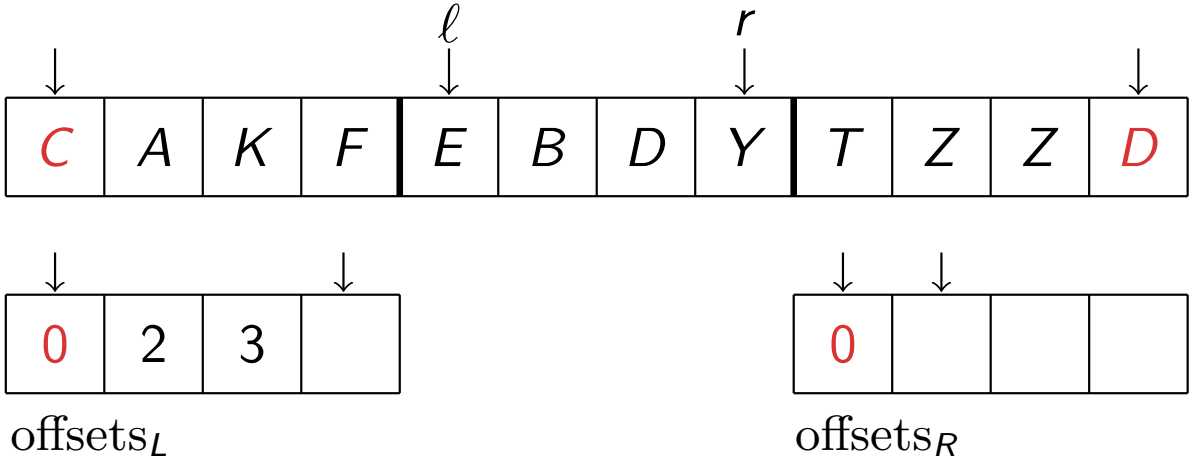


Block Partitioning

Choose block size B (we use $B = 128$)

1: **procedure** BLOCKPARTITION($A[\ell, \ldots, r]$, pivot)
2: **integer** $\text{offsets}_L[0, \ldots, B-1], \text{offsets}_R[0, \ldots, B-1]$
3: **integer** $\text{start}_L, \text{start}_R, \text{num}_L, \text{num}_R \leftarrow 0$
4: **while** $r - \ell + 1 > 2B$ **do** ▷ start main loop
5: ScanLeft
6: ScanRight
7: Rearrange ▷ swap elements
8: **end while** ▷ end main loop
9: scan and rearrange remaining elements
10: **end procedure**

Block size $B = 4$, pivot = "D":

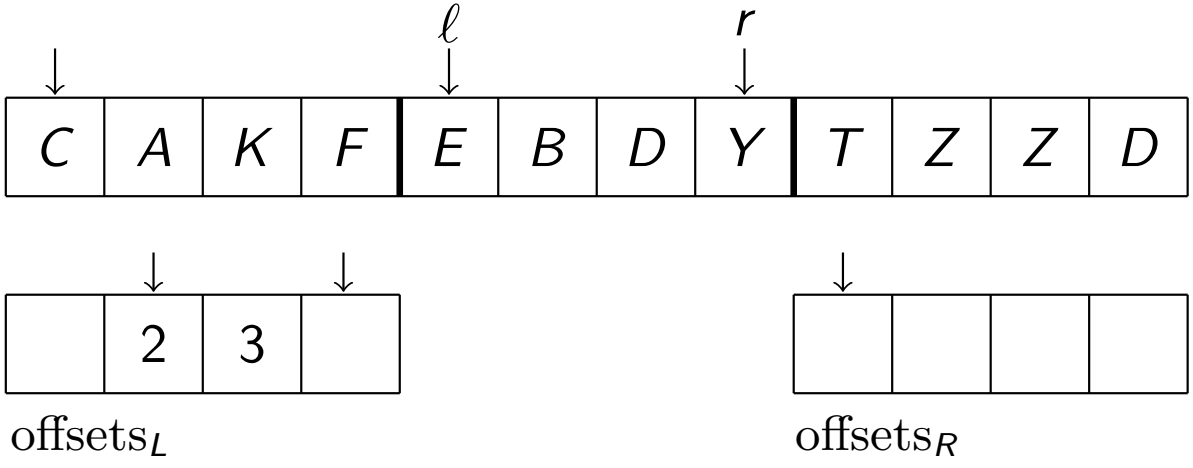


Block Partitioning

Choose block size B (we use $B = 128$)

```
 1: procedure BlockPartition(A[ℓ, ..., r], pivot)
 2:     integer offsets_L[0, ..., B − 1], offsets_R[0, ..., B − 1]
 3:     integer start_L, start_R, num_L, num_R ← 0
 4:     while r − ℓ + 1 > 2B do                          ▷ start main loop
 5:         ScanLeft
 6:         ScanRight
 7:         Rearrange                                    ▷ swap elements
 8:     end while                                        ▷ end main loop
 9:     scan and rearrange remaining elements
10: end procedure
```

Block size $B = 4$, pivot = "D":

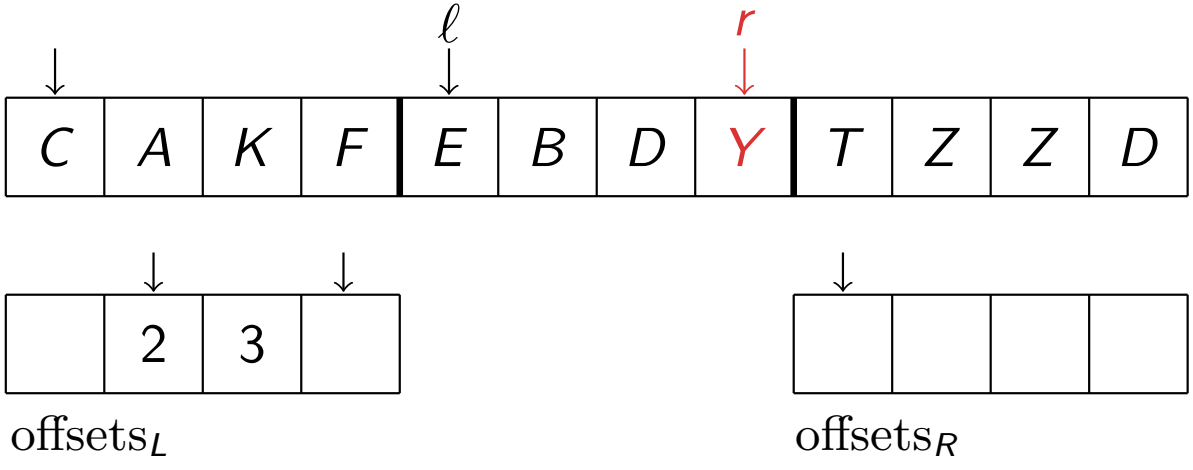


Block Partitioning

Choose block size B (we use $B = 128$)

```
 1: procedure BlockPartition(A[ℓ, ..., r], pivot)
 2:     integer offsets_L[0, ..., B − 1], offsets_R[0, ..., B − 1]
 3:     integer start_L, start_R, num_L, num_R ← 0
 4:     while r − ℓ + 1 > 2B do                          ▷ start main loop
 5:         ScanLeft
 6:         ScanRight
 7:         Rearrange                                    ▷ swap elements
 8:     end while                                        ▷ end main loop
 9:     scan and rearrange remaining elements
10: end procedure
```

Block size $B = 4$, pivot = "D":

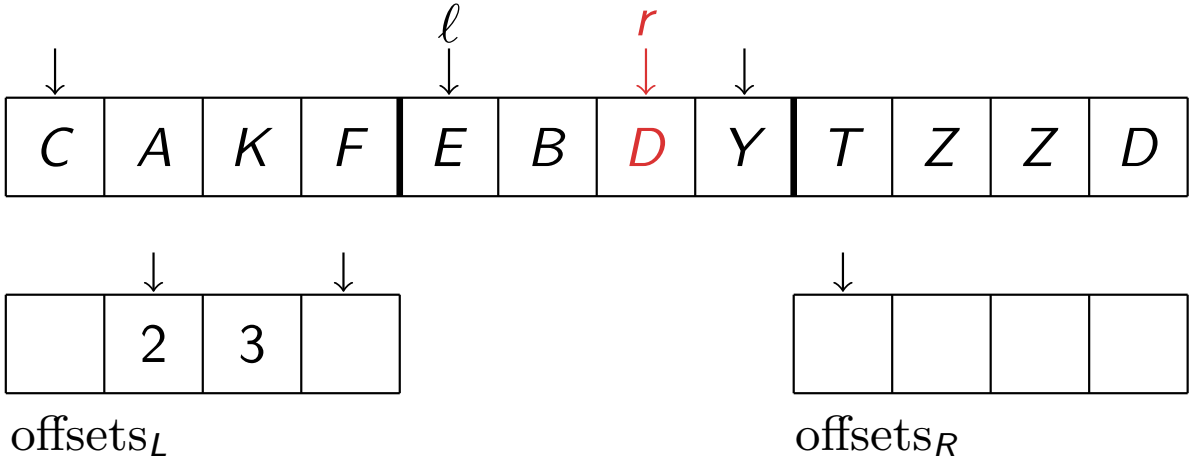


Block Partitioning

Choose block size B (we use $B = 128$)

1: **procedure** BLOCKPARTITION($A[\ell, \ldots, r]$, pivot)
2: **integer** $\text{offsets}_L[0, \ldots, B-1], \text{offsets}_R[0, \ldots, B-1]$
3: **integer** $\text{start}_L, \text{start}_R, \text{num}_L, \text{num}_R \leftarrow 0$
4: **while** $r - \ell + 1 > 2B$ **do** ▷ start main loop
5: ScanLeft
6: ScanRight
7: Rearrange ▷ swap elements
8: **end while** ▷ end main loop
9: scan and rearrange remaining elements
10: **end procedure**

Block size $B = 4$, pivot = "D":

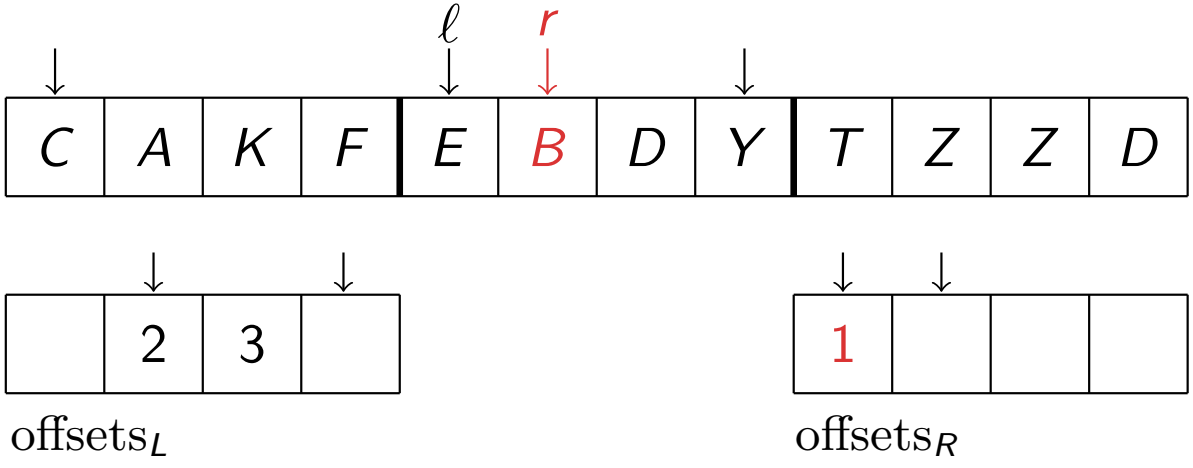


Block Partitioning

Choose block size B (we use $B = 128$)

1: **procedure** BLOCKPARTITION($A[\ell, \ldots, r]$, pivot)
2: **integer** $\text{offsets}_L[0, \ldots, B-1], \text{offsets}_R[0, \ldots, B-1]$
3: **integer** $\text{start}_L, \text{start}_R, \text{num}_L, \text{num}_R \leftarrow 0$
4: **while** $r - \ell + 1 > 2B$ **do** ▷ start main loop
5: ScanLeft
6: ScanRight
7: Rearrange ▷ swap elements
8: **end while** ▷ end main loop
9: scan and rearrange remaining elements
10: **end procedure**

Block size $B = 4$, pivot = "D":

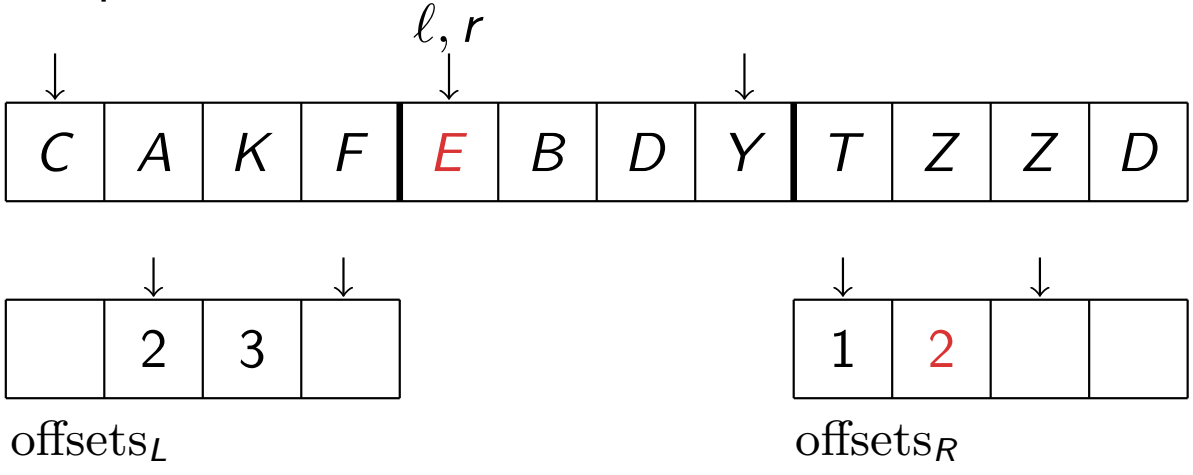


Block Partitioning

Choose block size B (we use $B = 128$)

```
 1: procedure BlockPartition(A[ℓ, ..., r], pivot)
 2:     integer offsets_L[0, ..., B − 1], offsets_R[0, ..., B − 1]
 3:     integer start_L, start_R, num_L, num_R ← 0
 4:     while r − ℓ + 1 > 2B do                          ▷ start main loop
 5:         ScanLeft
 6:         ScanRight
 7:         Rearrange                                    ▷ swap elements
 8:     end while                                        ▷ end main loop
 9:     scan and rearrange remaining elements
10: end procedure
```

Block size $B = 4$, pivot = "D":

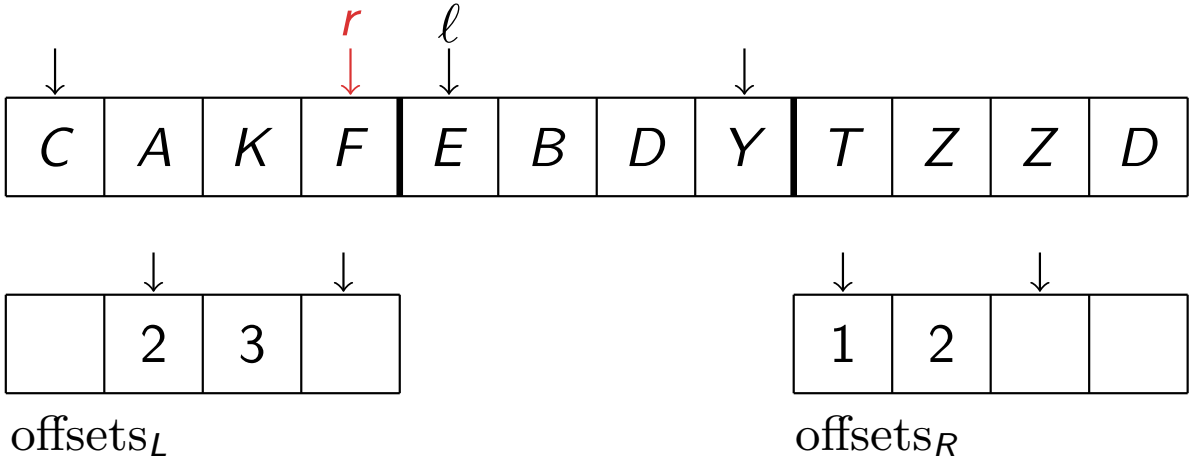


Block Partitioning

Choose block size B (we use $B = 128$)

```
 1: procedure BlockPartition(A[ℓ, ..., r], pivot)
 2:     integer offsets_L[0, ..., B − 1], offsets_R[0, ..., B − 1]
 3:     integer start_L, start_R, num_L, num_R ← 0
 4:     while r − ℓ + 1 > 2B do                          ▷ start main loop
 5:         ScanLeft
 6:         ScanRight
 7:         Rearrange                                    ▷ swap elements
 8:     end while                                        ▷ end main loop
 9:     scan and rearrange remaining elements
10: end procedure
```

Block size $B = 4$, pivot = "D":

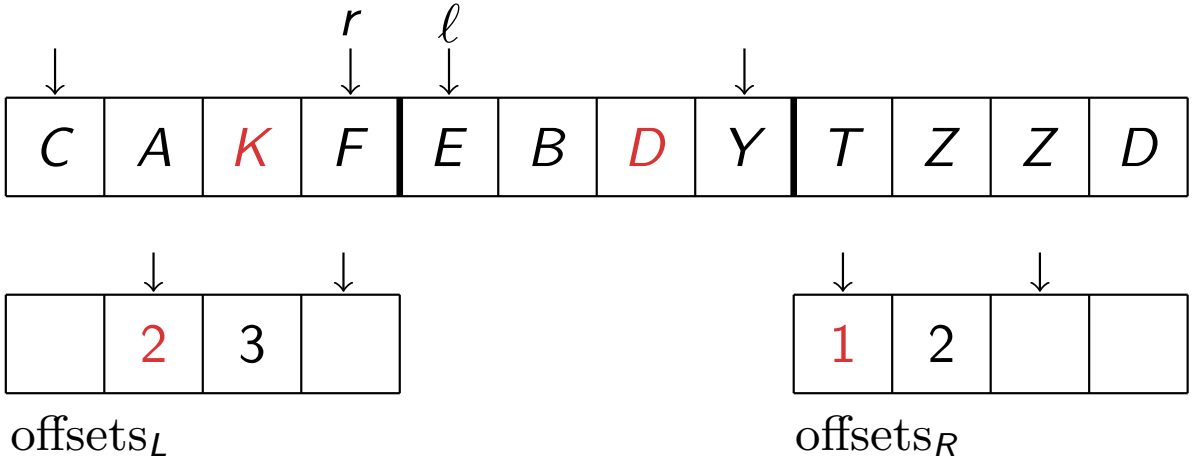


Block Partitioning

Choose block size B (we use $B = 128$)

```
 1: procedure BlockPartition(A[ℓ, ..., r], pivot)
 2:     integer offsets_L[0, ..., B − 1], offsets_R[0, ..., B − 1]
 3:     integer start_L, start_R, num_L, num_R ← 0
 4:     while r − ℓ + 1 > 2B do                          ▷ start main loop
 5:         ScanLeft
 6:         ScanRight
 7:         Rearrange                                    ▷ swap elements
 8:     end while                                        ▷ end main loop
 9:     scan and rearrange remaining elements
10: end procedure
```

Block size $B = 4$, pivot = "D":

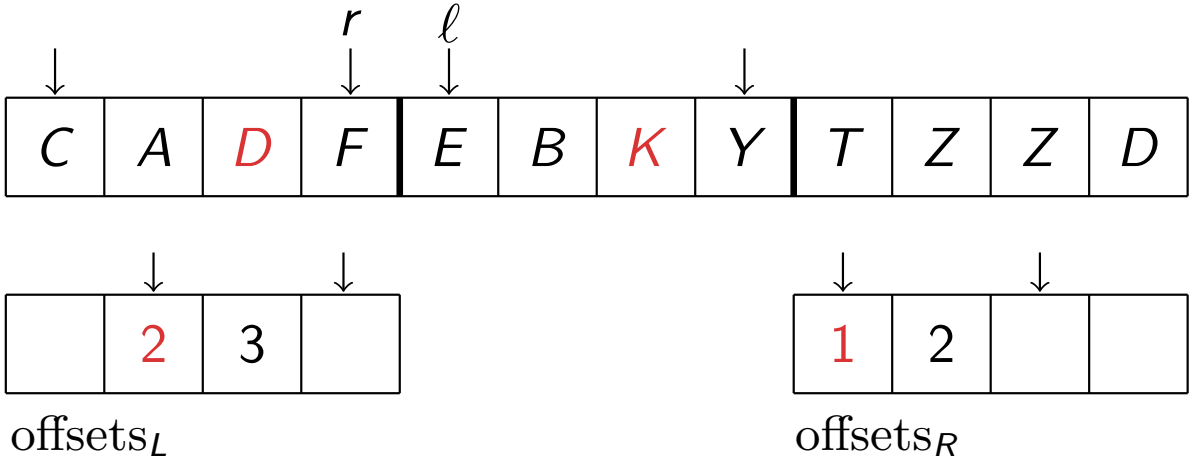


Block Partitioning

Choose block size B (we use $B = 128$)

1: **procedure** BLOCKPARTITION($A[\ell, \ldots, r]$, pivot)
2: **integer** $\text{offsets}_L[0, \ldots, B-1], \text{offsets}_R[0, \ldots, B-1]$
3: **integer** $\text{start}_L, \text{start}_R, \text{num}_L, \text{num}_R \leftarrow 0$
4: **while** $r - \ell + 1 > 2B$ **do** ▷ start main loop
5: ScanLeft
6: ScanRight
7: Rearrange ▷ swap elements
8: **end while** ▷ end main loop
9: scan and rearrange remaining elements
10: **end procedure**

Block size $B = 4$, pivot = "D":

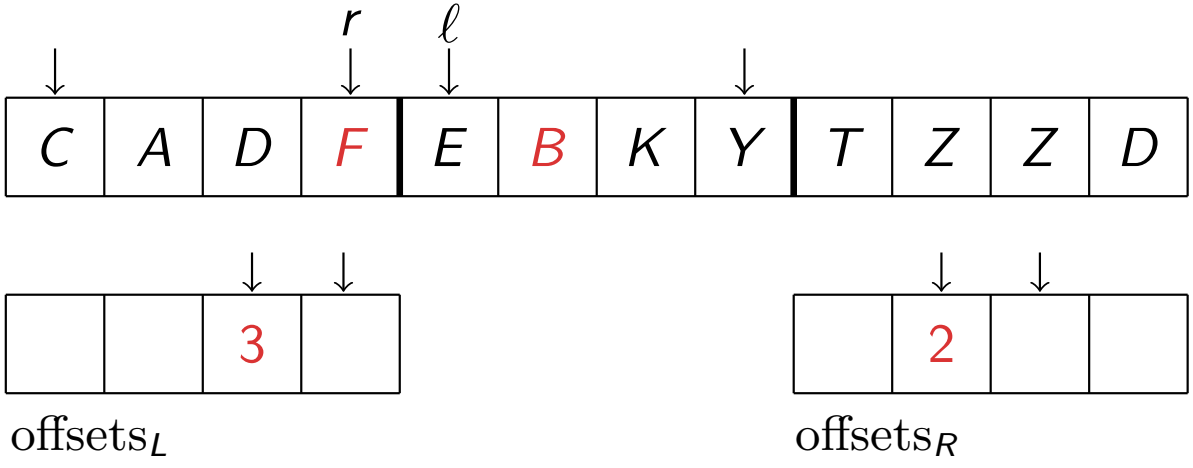


Block Partitioning

Choose block size B (we use $B = 128$)

1: **procedure** BLOCKPARTITION($A[\ell, \ldots, r]$, pivot)
2: **integer** $\text{offsets}_L[0, \ldots, B-1], \text{offsets}_R[0, \ldots, B-1]$
3: **integer** $\text{start}_L, \text{start}_R, \text{num}_L, \text{num}_R \leftarrow 0$
4: **while** $r - \ell + 1 > 2B$ **do** ▷ start main loop
5: ScanLeft
6: ScanRight
7: Rearrange ▷ swap elements
8: **end while** ▷ end main loop
9: scan and rearrange remaining elements
10: **end procedure**

Block size $B = 4$, pivot = "D":

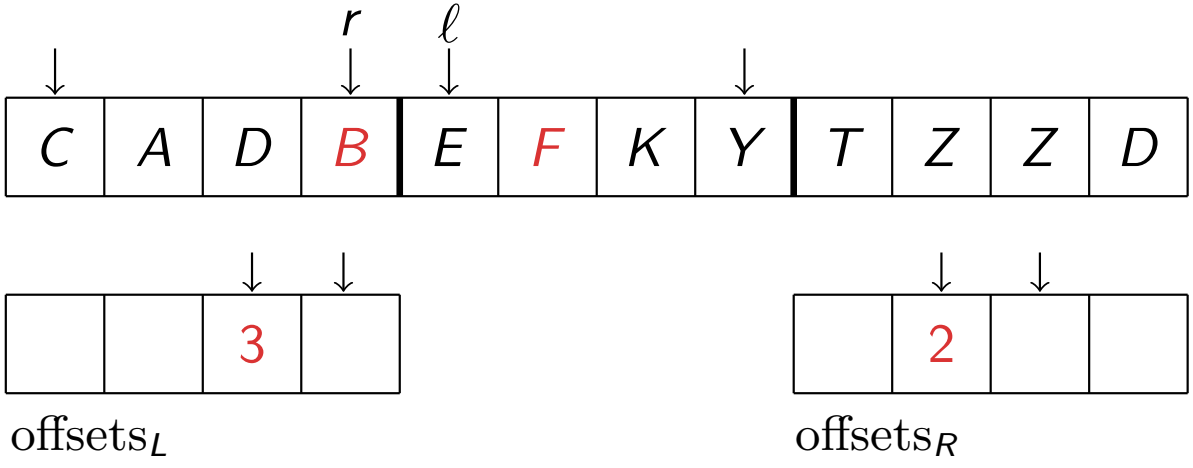


Block Partitioning

Choose block size B (we use $B = 128$)

1: **procedure** BLOCKPARTITION($A[\ell, \ldots, r]$, pivot)
2: **integer** $\text{offsets}_L[0, \ldots, B-1], \text{offsets}_R[0, \ldots, B-1]$
3: **integer** $\text{start}_L, \text{start}_R, \text{num}_L, \text{num}_R \leftarrow 0$
4: **while** $r - \ell + 1 > 2B$ **do** ▷ start main loop
5: ScanLeft
6: ScanRight
7: Rearrange ▷ swap elements
8: **end while** ▷ end main loop
9: scan and rearrange remaining elements
10: **end procedure**

Block size $B = 4$, pivot = "D":

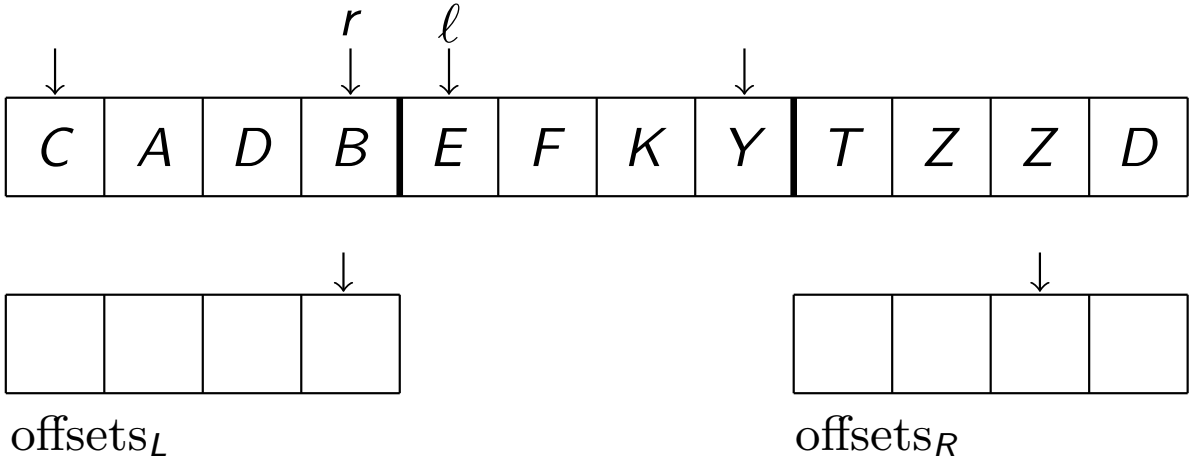


Block Partitioning

Choose block size B (we use $B = 128$)

```
 1: procedure BlockPartition(A[ℓ, ..., r], pivot)
 2:     integer offsets_L[0, ..., B − 1], offsets_R[0, ..., B − 1]
 3:     integer start_L, start_R, num_L, num_R ← 0
 4:     while r − ℓ + 1 > 2B do                          ▷ start main loop
 5:         ScanLeft
 6:         ScanRight
 7:         Rearrange                                    ▷ swap elements
 8:     end while                                        ▷ end main loop
 9:     scan and rearrange remaining elements
10: end procedure
```

Block size $B = 4$, pivot = "D":

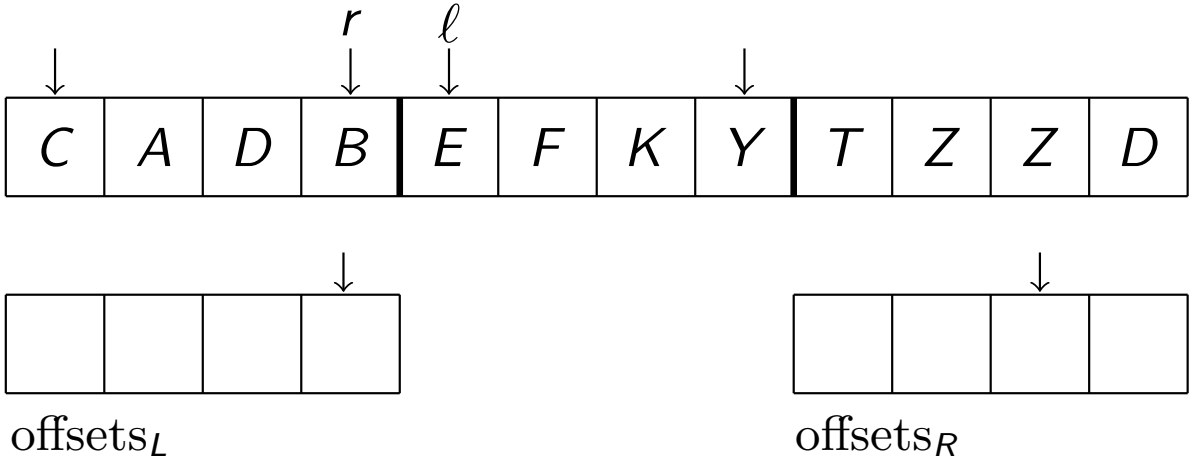


Block Partitioning

Choose block size B (we use $B = 128$)

```
 1: procedure BlockPartition(A[ℓ, ..., r], pivot)
 2:     integer offsets_L[0, ..., B − 1], offsets_R[0, ..., B − 1]
 3:     integer start_L, start_R, num_L, num_R ← 0
 4:     while r − ℓ + 1 > 2B do                          ▷ start main loop
 5:         ScanLeft
 6:         ScanRight
 7:         Rearrange                                    ▷ swap elements
 8:     end while                                        ▷ end main loop
 9:     scan and rearrange remaining elements
10: end procedure
```

Block size $B = 4$, pivot = "D":



Block Partitioning

```
1: procedure ScanLeft
2:     if num_L = 0 then                          ▷ if left buffer is empty, refill it
3:         start_L ← 0
4:         for i = 0, ..., B − 1 do
5:             offsets_L[num_L] ← i
6:             num_L += (pivot ≥ A[ℓ + i])        ▷ comparison returns 0 or 1
7:         end for
8:     end if
9: end procedure
```

- Current offset is always written into offset array – without considering outcome of the comparison.
- Conversion from Boolean (processor flag) to integer – supported in hardware on x86 and many other processors (`setcc`)

$\rightsquigarrow$ (almost) no unpredictable conditional statements

Block Partitioning

- Number of comparisons, moves,. . . like for classical Quicksort.

Block Partitioning

- Number of comparisons, moves,. . . like for classical Quicksort.
- At most twice as many data accesses.

Block Partitioning

- Number of comparisons, moves,. . . like for classical Quicksort.
- At most twice as many data accesses.
  But: most additional accesses go to L1 cache.

Block Partitioning

- Number of comparisons, moves,. . . like for classical Quicksort.
- At most twice as many data accesses.
  But: most additional accesses go to L1 cache.

Theorem

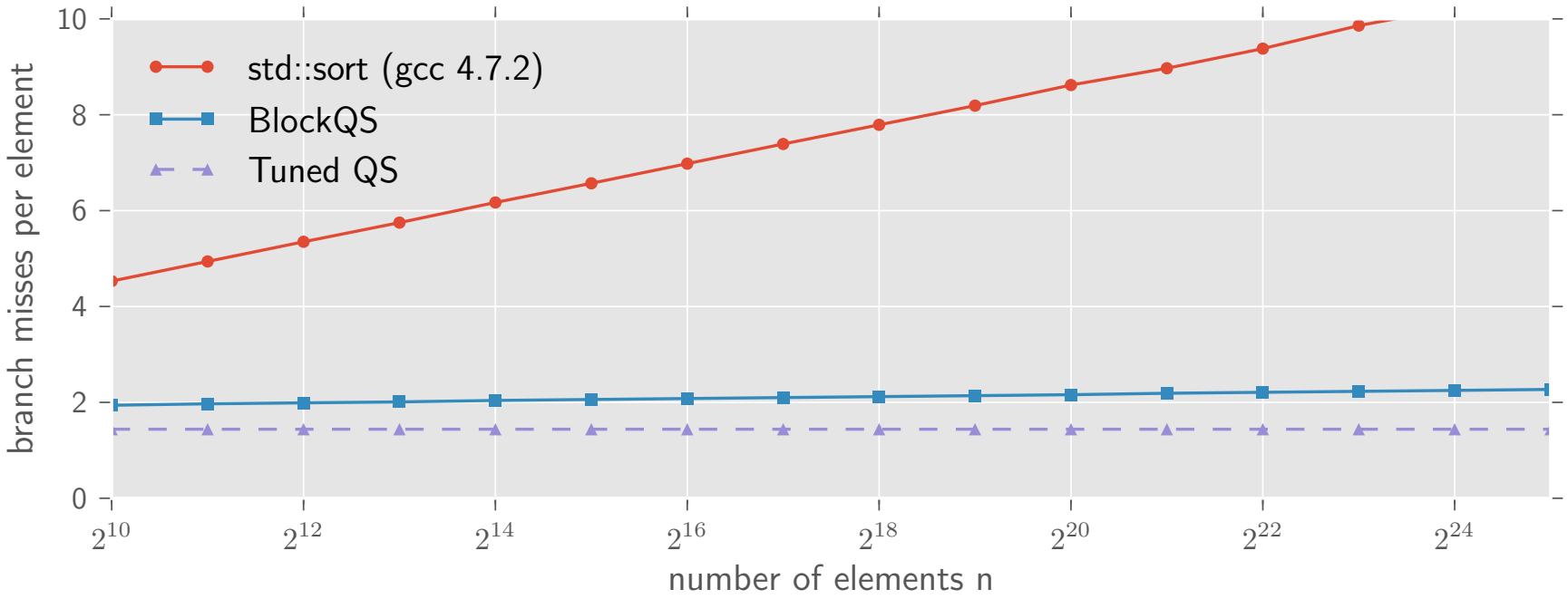
BlockQuicksort with median-of-three induces less than $\frac{8}{B} n \log n + \mathcal{O}(n)$ *branch mispredictions on average with static branch prediction.*

Block Partitioning

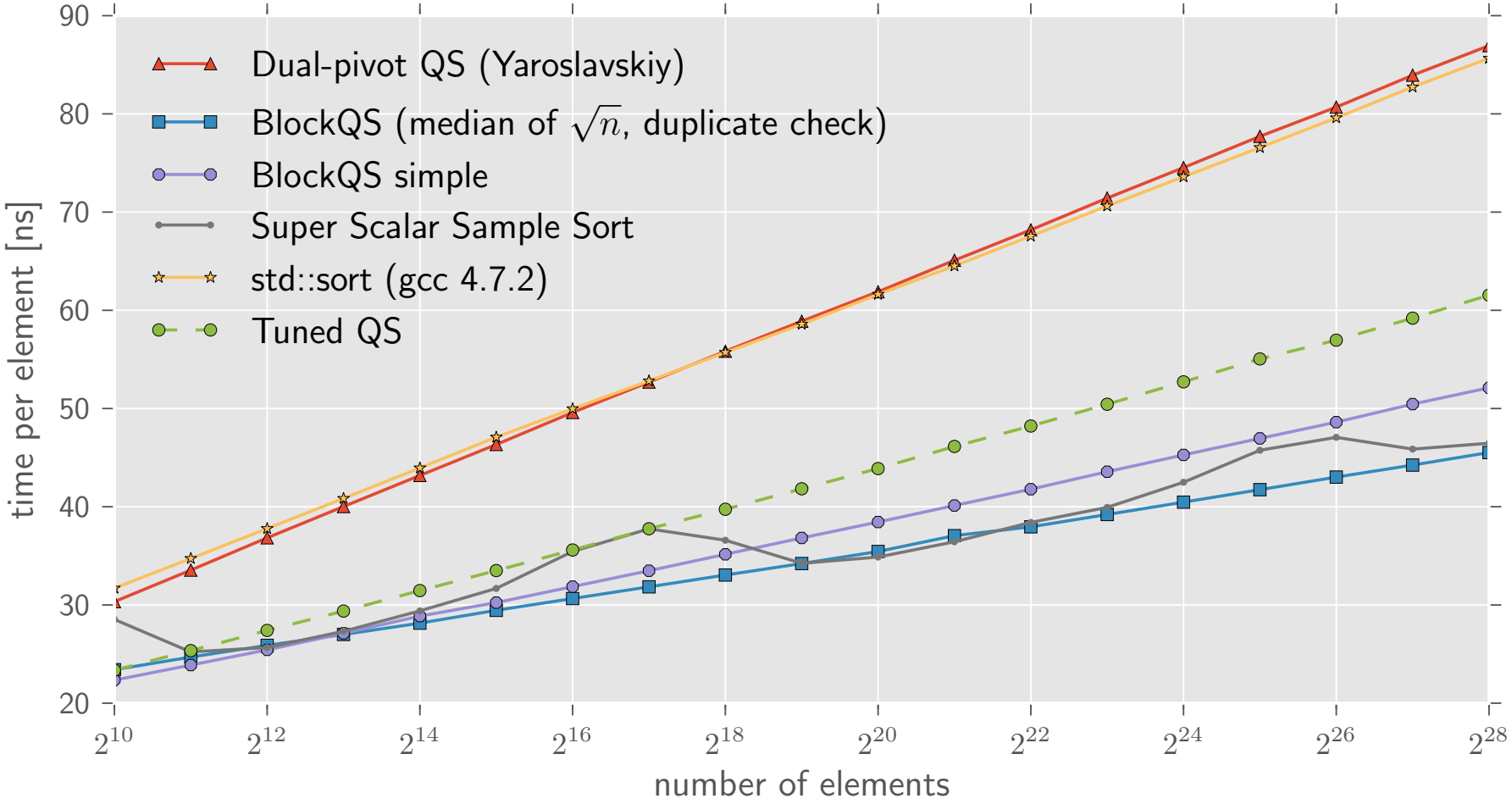
- Number of comparisons, moves,. . . like for classical Quicksort.
- At most twice as many data accesses.
  But: most additional accesses go to L1 cache.

Theorem

BlockQuicksort with median-of-three induces less than $\frac{8}{B} n \log n + \mathcal{O}(n)$ *branch mispredictions on average with static branch prediction.*



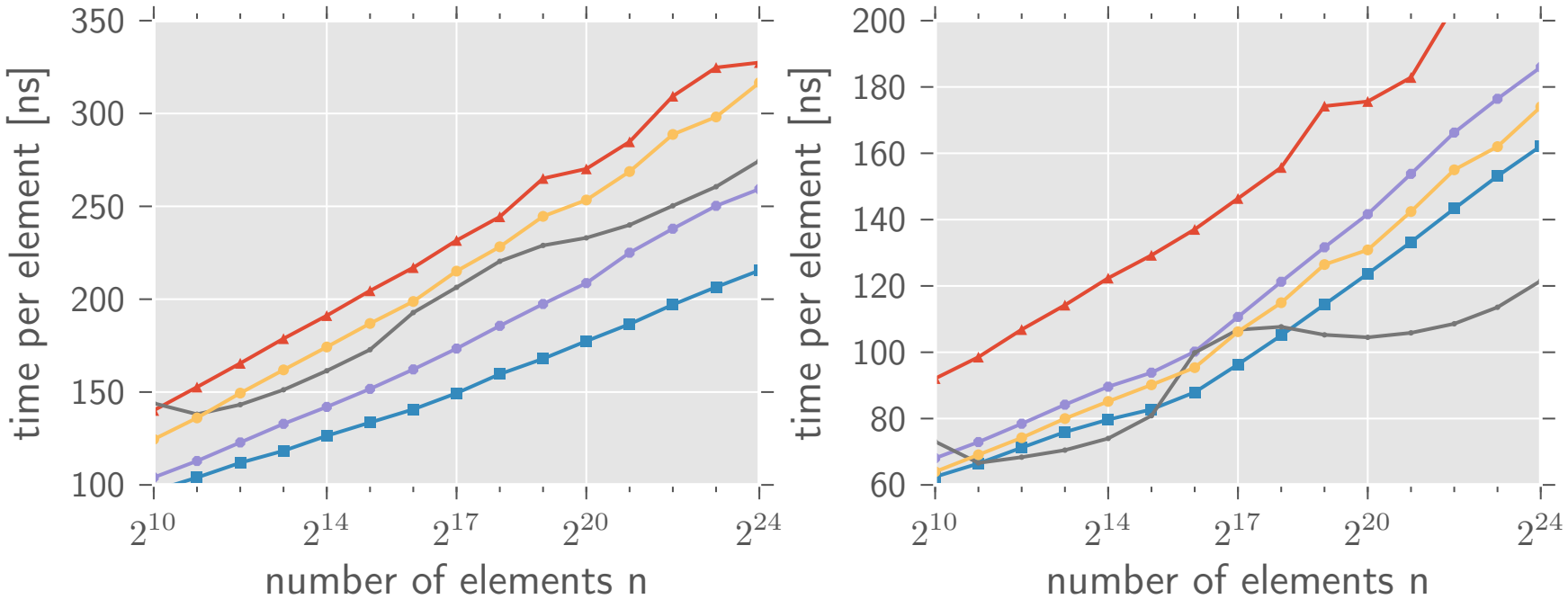
Experiments with random permutations of integers



Test environment:

- Intel Core i5-2500K CPU (3.30GHz) with 16GB RAM
- Ubuntu Linux 64bit version 14.04.4
- g++ (4.8.4) compiler with flags `-O3 -march=native`

Experiments with other data types



left Record of 10 `doubles`, comparison via the Euclidean norm.

right Record of 21 `ints`, only the first component is compared.