



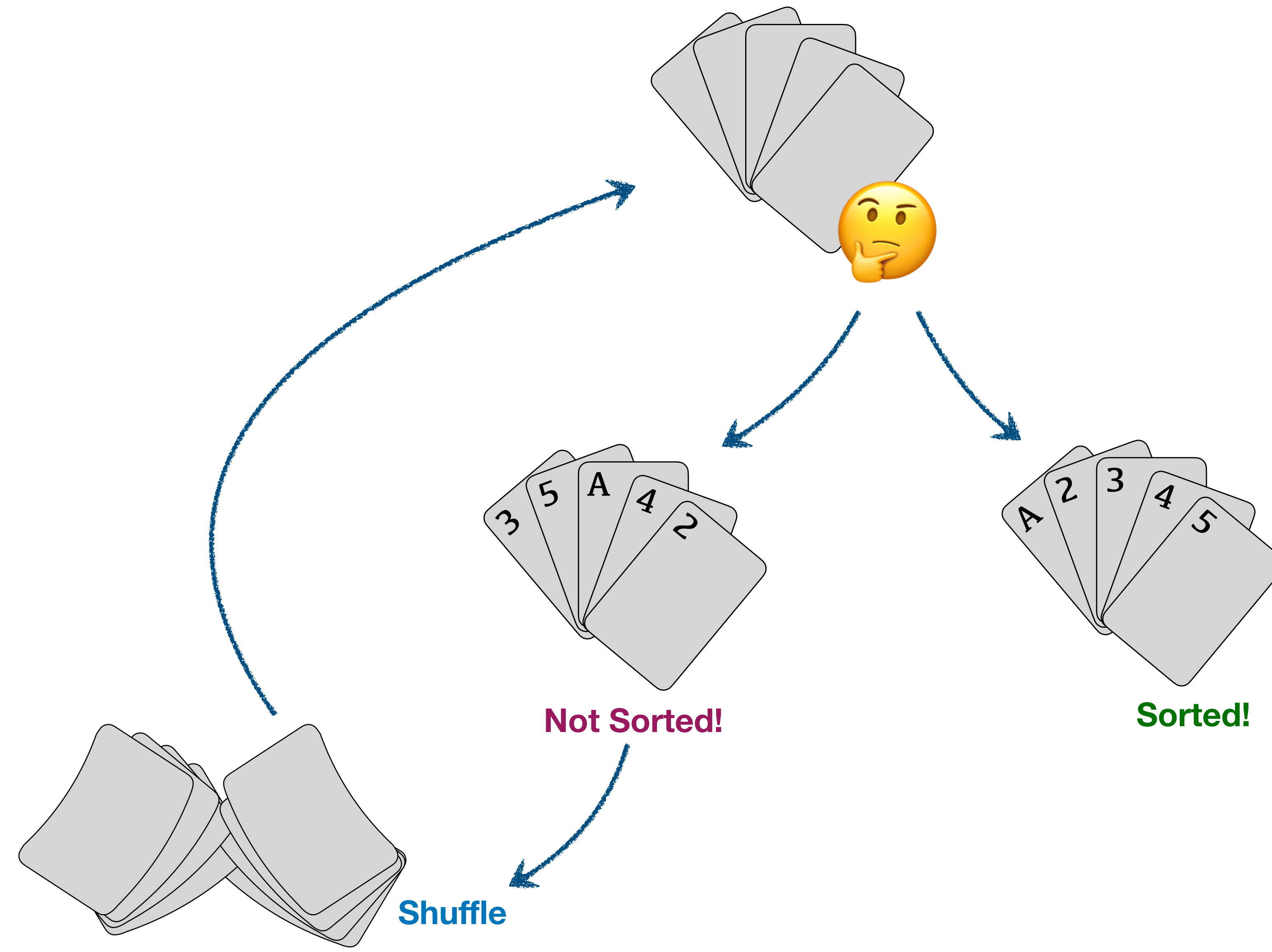
排序 Sorting

钮鑫涛 王智彬
Nanjing University
2025 Fall

The slides are mainly adapted from the original ones shared by Chaodong Zheng and Kevin Wayne. Thanks for their supports! We also use some materials from stanford-cs161.



Bogosort: The stupid sort



My Bogosort, when it doesn't sort the array correctly the thousands time





The Sorting Problem

- Sort n numbers into ascending order.
- We can actually sort a collection of any type of data, as long as a **total order** is defined for that type of data.
- That is, for any distinct data items a and b , we compare them, i.e., we can determine:
 - $a < b$, or $b < a$, otherwise, $a = b$, where “ $<$ ” is a binary relation:
 - E.g., in Java, to use `Collections.sort(List<DataType> list, Comparator<DataType> comparator)` for sorting, you should implement the `comparator` and define the following function in it:

```
public int compare(DataType item1, DataType item2)
```
- We can also sort partially ordered items (more on this later).



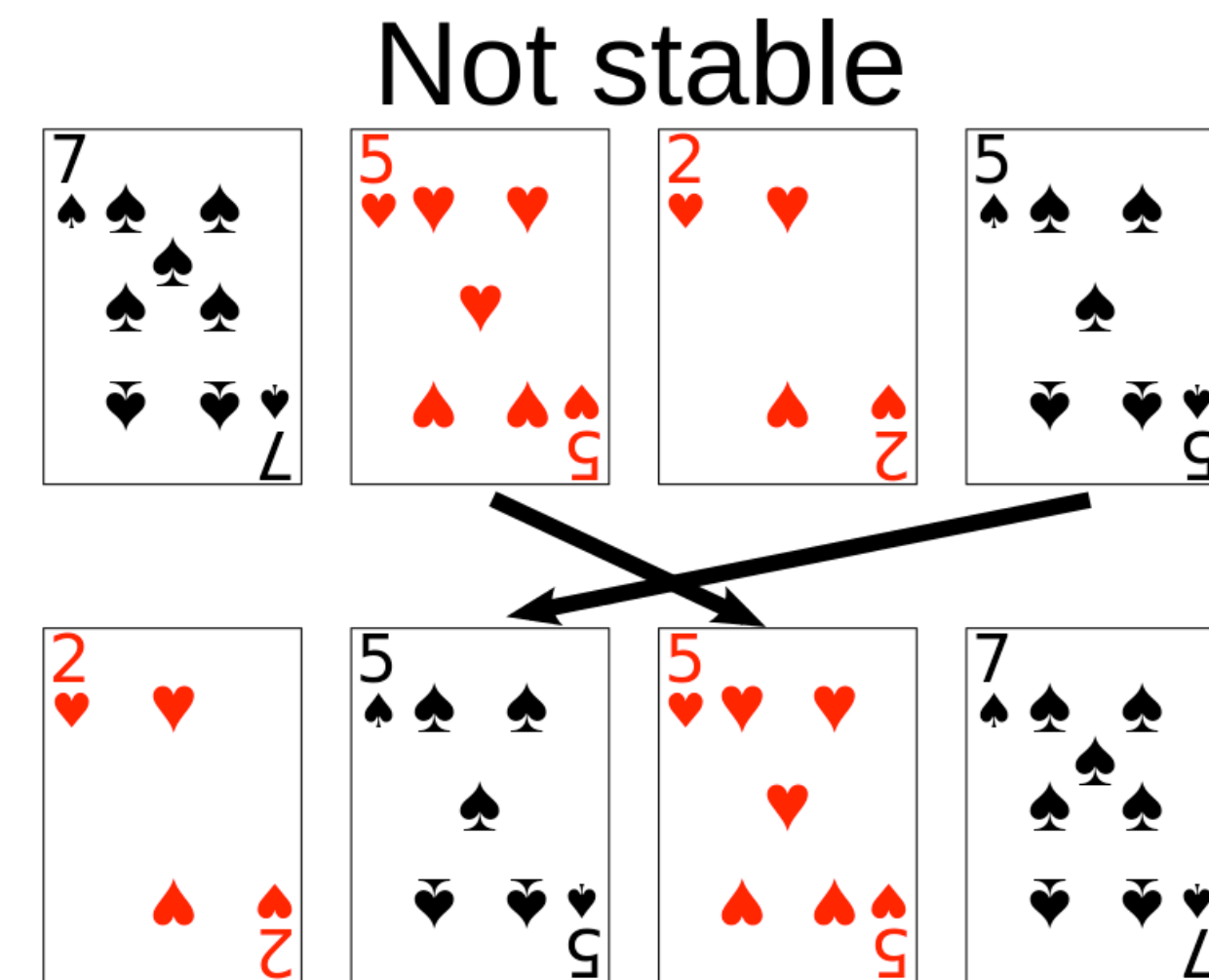
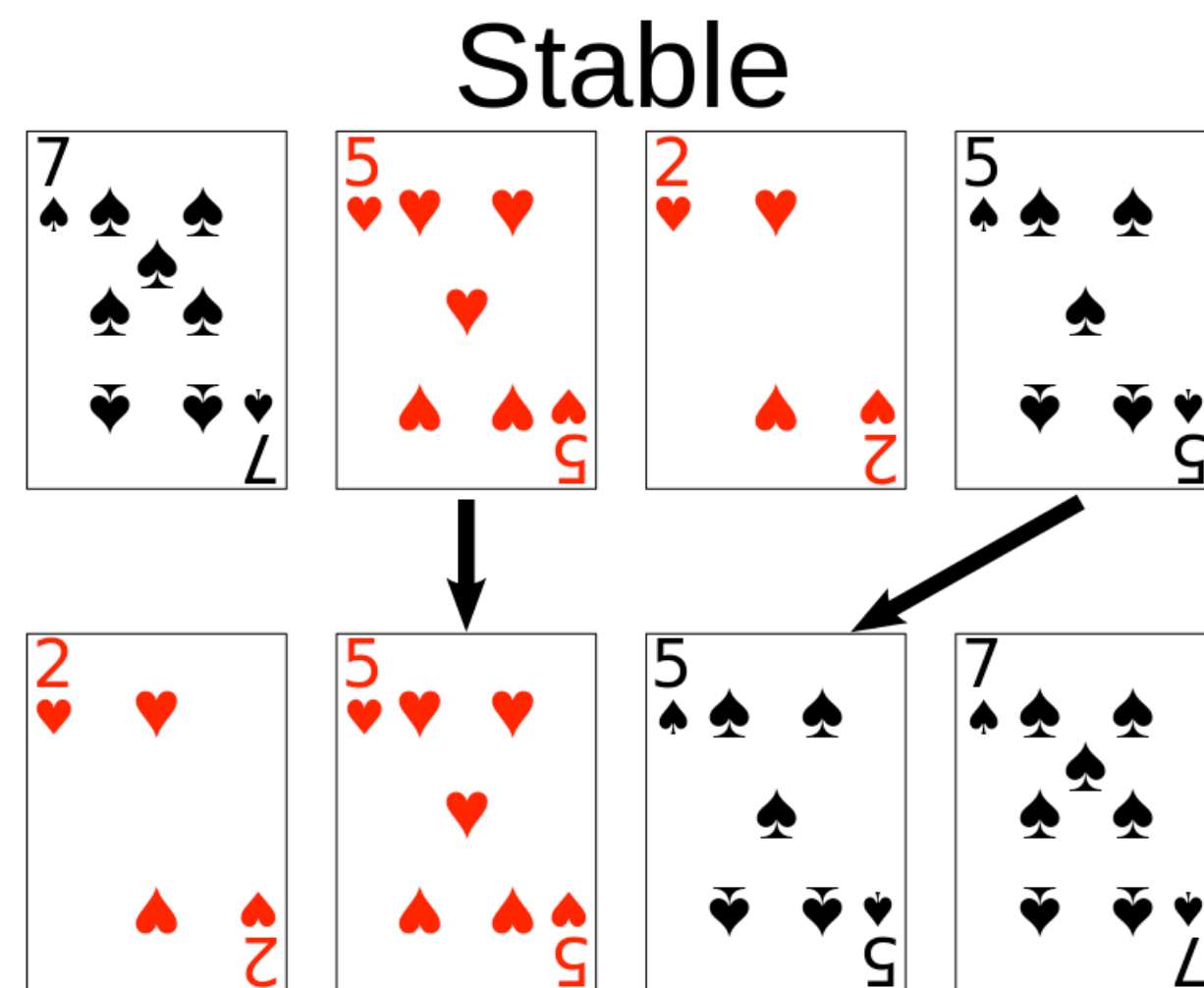
Sorting algorithms till now

- **Insertion Sort:** gradually increase size of sorted part.
 - $O(n^2)$ time, $O(1)$ space
- **Merge Sort:** example of divide-and-conquer
 - $O(n \log n)$ time, $O(n)$ space
- **Heap Sort:** leverage the heap data structure
 - $O(n \log n)$ time, $O(1)$ space



Characteristics of sorting algorithms

- In-place (原地): a sorting algorithm is in-place if $O(1)$ extra space is needed beyond input.
- Stability (稳定): a sorting algorithm is stable if numbers with the same value appear in the output array in the same order as they do in the input array.





Sorting algorithms till now

Insertion Sort: gradually increase size of sorted part.

- ▶ $O(n^2)$ time, $O(1)$ space
- ▶ In-place, and stable.

Merge Sort: example of divide-and-conquer

- ▶ $O(n \log n)$ time, $O(n)$ space
- ▶ Not in-place, but stable.

Heap Sort: leverage the heap data structure

- ▶ $O(n \log n)$ time, $O(1)$ space
- ▶ In-place, but not stable.

Counterexample for stability: $\langle 2a, 2b, 1 \rangle$:
It is already a max heap, then

1. $2a$ is extracted, and placed in the end
2. $2b$ is extracted, and placed in the end but one index

At last, we get $\langle 1, 2b, 2a \rangle$



Elementary sorting



The Selection Sort Algorithm

- **Basic idea:** pick out minimum element from input, then recursively sort remaining elements, and finally concatenate the minimum element with sorted remaining elements.

SelectionSortRec(A):

```
if  $|A| = 1$   
    return  $A$   
else  
     $min := \text{GetMinElement}(A)$   
     $A' := \text{RemoveElement}(A, min)$   
    return  $\text{Concatenate}(min, \text{SelectionSortRec}(A'))$ 
```

SelectionSort(A):

```
for  $i := 1$  to  $A.length$   
     $minIdx := i$   
    for  $j := i + 1$  to  $A.length$   
        if  $A[j] < A[minIdx]$   
             $minIdx := j$   
     $\text{Swap}(i, minIdx)$ 
```



Analysis of SelectionSort

- Why it is correct? (What is the loop invariant?)
 - After the i^{th} iteration, the first i items are sorted, and they are the i smallest elements in the original array.
- Time complexity for sorting n items?

$$\sum_{i=1}^{n-1} (\Theta(1) + \Theta(n - i)) = \Theta(n^2)$$

SelectionSort(A):

```
for  $i := 1$  to  $A.length$ 
     $minIdx := i$ 
    for  $j := i + 1$  to  $A.length$ 
        if  $A[j] < A[minIdx]$ 
             $minIdx := j$ 
    Swap( $i, minIdx$ )
```




Analysis of SelectionSort

- Space complexity?
 - $O(1)$ extra space, thus in-place
- Stability?
 - Not stable! Swap operation can mess up relative order
 - Counterexample for stability: $\langle 2a, 2b, 1 \rangle$

SelectionSort(A):

```
for  $i := 1$  to  $A.length$   
     $minIdx := i$   
    for  $j := i + 1$  to  $A.length$   
        if  $A[j] < A[minIdx]$   
             $minIdx := j$   
     $Swap(i, minIdx)$ 
```



Before we move on

SelectionSort(A):

```
for i := 1 to A.length  
  minIdx := i  
  for j := i + 1 to A.length  
    if A[j] < A[minIdx]  
      minIdx := j  
  Swap(i, minIdx)
```

SelectionSortRec(A):

```
if |A| = 1  
  return A  
else  
  min := GetMinElement(A)  
  A' := RemoveElement(A, min)  
  return Concatenate(min, SelectionSortRec(A'))
```

Get the minimal element and extract it?

Similar operations: **HeapGetMax**, **HeapExtractMax**



Before we move on

SelectionSortRec(A):

```
if  $|A| = 1$   
    return  $A$   
else  
     $min := GetMinElement(A)$   
     $A' := RemoveElement(A, min)$   
    return Concatenate( $min, SelectionSortRec(A')$ )
```

SelectionSortRecVariant(A):

```
if  $|A| = 1$   
    return  $A$   
else  
     $max := GetMaxElement(A)$   
     $A' := RemoveElement(A, max)$   
    return Concatenate( $SelectionSortRec(A'), max$ )
```

Let A get organized as a heap, then it leads to the faster HeapSort algorithm.

The choice of data structure affects the performance of algorithms!



The Bubble Sort Algorithm

- **Basic idea:** repeatedly step through the array, compare adjacent pairs and swaps them if they are in the wrong order. Thus, larger elements "bubble" to the "top".

BubbleSort(A):

```
for  $i := A.length$  down to 2  
  for  $j := 1$  to  $i - 1$   
    if  $A[j] > A[j+1]$   
      Swap( $A[j]$ ,  $A[j+1]$ )
```





Analysis of BubbleSort

- Correctness:
 - What is the invariant?
- Time complexity:
 - $\Theta(n^2)$
- Space complexity:
 - $\Theta(1)$
- Stability:
 - Stable

BubbleSort(A):

```
for  $i := A.length$  down to 2  
    for  $j := 1$  to  $i - 1$   
        if  $A[j] > A[j+1]$   
            Swap( $A[j]$ ,  $A[j+1]$ )
```



Improving BubbleSort

BubbleSort(A):

for $i := A.length$ **down to** 2

for $j := 1$ **to** $i - 1$

if $A[j] > A[j+1]$

 Swap($A[j]$, $A[j+1]$)

- What if in one iteration we never swap data items?
 - Then $A[1...i]$ are sorted, and we are done! (Why?)



Improving BubbleSort

BubbleSortImporved(A):

$n := A.length$

repeat

$swapped := false$

for $j := 1$ **to** $n - 1$

if $A[j] > A[j+1]$

 Swap($A[j]$, $A[j+1]$)

$swapped := true$

$n := n - 1$

until $swapped = false$

- When the input is mostly sorted, this variant performs much better.
 - Particularly, when the input is sorted, this variant has $O(n)$ runtime.
 - Other algorithms that also have this property, E.g., InsertionSort.
 - Nonetheless, the worst case performance is still $\Theta(n^2)$.
 - E.g., when input is reversely sorted.



Improving BubbleSort

BubbleSortImporved(A):

$n := A.length$

repeat

$swapped := false$

for $j := 1$ **to** $n - 1$

if $A[j] > A[j+1]$

 Swap($A[j]$, $A[j+1]$)

$swapped := true$

$n := n - 1$

until $swapped = false$

$n = 5$

3	2	1	8	9	12	15
---	---	---	---	---	----	----

Swap

2	3	1	8	9	12	15
---	---	---	---	---	----	----

Swap

2	1	3	8	9	12	15
---	---	---	---	---	----	----

No Swap

2	1	3	8	9	12	15
---	---	---	---	---	----	----

No Swap

2	1	3	8	9	12	15
---	---	---	---	---	----	----

$swapped = true$



Improving BubbleSort

BubbleSortImproved(A):

$n := A.length$

repeat

$swapped := false$

for $j := 1$ **to** $n - 1$

if $A[j] > A[j+1]$

$Swap(A[j], A[j+1])$

$swapped := true$

$n := n - 1$

until $swapped = false$

$n = 4$

2	1	3	8	9	12	15
---	---	---	---	---	----	----

Swap

1	2	3	8	9	12	15
---	---	---	---	---	----	----

No Swap

1	2	3	8	9	12	15
---	---	---	---	---	----	----

No Swap

1	2	3	8	9	12	15
---	---	---	---	---	----	----

$swapped = true$



Improving BubbleSort

BubbleSortImporved(A):

$n := A.length$

repeat

$swapped := false$

for $j := 1$ **to** $n - 1$

if $A[j] > A[j+1]$

 Swap($A[j]$, $A[j+1]$)

$swapped := true$

$n := n - 1$

until $swapped = false$

$n = 3$

1	2	3	8	9	12	15
---	---	---	---	---	----	----

No Swap

1	2	3	8	9	12	15
---	---	---	---	---	----	----

No Swap

1	2	3	8	9	12	15
---	---	---	---	---	----	----

$swapped = false$



Improving BubbleSort

BubbleSortImproved(A):

$n := A.length$

repeat

$swapped := false$

for $j := 1$ **to** $n - 1$

if $A[j] > A[j+1]$

Swap($A[j]$, $A[j+1]$)

$swapped := true$

$n := n - 1$

until $swapped = false$

$n = 5$

3	2	1	8	9	12	15
---	---	---	---	---	----	----

Swap

2	3	1	8	9	12	15
---	---	---	---	---	----	----

Swap

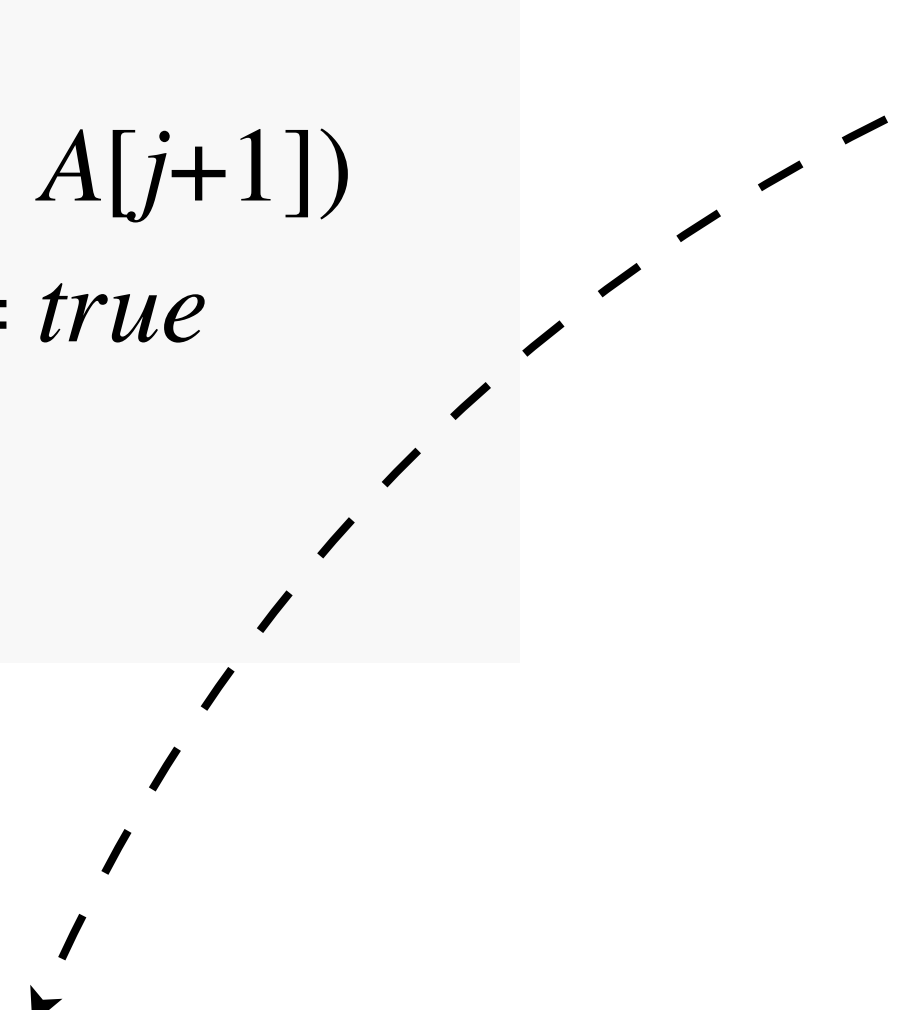
2	1	3	8	9	12	15
---	---	---	---	---	----	----

No Swap

2	1	3	8	9	12	15
---	---	---	---	---	----	----

No Swap

2	1	3	8	9	12	15
---	---	---	---	---	----	----



The last swap index is 2, and then the following items has no swap, indicating that the following items are already sorted!



Improving BubbleSort

BubbleSortImproved(A):

$n := A.length$

repeat

$swapped := false$

for $j := 1$ **to** $n - 1$

if $A[j] > A[j+1]$

$Swap(A[j], A[j+1])$

$swapped := true$

$n := n - 1$

until $swapped = false$

$n = 5$

3	2	1	8	9	12	15
---	---	---	---	---	----	----

Swap

2	3	1	8	9	12	15
---	---	---	---	---	----	----

Swap

2	1	3	8	9	12	15
---	---	---	---	---	----	----

No Swap

2	1	3	8	9	12	15
---	---	---	---	---	----	----

No Swap

2	1	3	8	9	12	15
---	---	---	---	---	----	----

The last swap index is 2, and then the following items has no swap, indicating that the following items are already sorted!

Therefore, in the next step, n should be 2



Improving BubbleSort

- We can be more aggressive when reducing n after each iteration: in $A[1 \dots n]$, items after the last swap are all in correct sorted position.

BubbleSortImporved(A):

```
n := A.length
repeat
  swapped := false
  for j := 1 to n - 1
    if A[j] > A[j+1]
      Swap(A[j], A[j+1])
      swapped := true
  n := n - 1
until swapped = false
```



BubbleSortImporvedFurther(A):

```
n := A.length
repeat
  lastSwapIdx := -1
  for j := 1 to n - 1
    if A[j] > A[j+1]
      Swap(A[j], A[j+1])
      lastSwapIdx := j + 1
  n := lastSwapIdx - 1
until n <= 1
```



Improving BubbleSort

BubbleSortImporvedFurther(A):

$n := A.length$

repeat

$lastSwapIdx := -1$

for $j := 1$ **to** $n - 1$

if $A[j] > A[j+1]$

$Swap(A[j], A[j+1])$

$lastSwapIdx := j + 1$

$n := lastSwapIdx - 1$

until $n \leq 1$

$n = 5$

3	2	1	8	9	12	15
---	---	---	---	---	----	----

$lastSwapIdx = 2$

2	3	1	8	9	12	15
---	---	---	---	---	----	----

$lastSwapIdx = 3$

2	1	3	8	9	12	15
---	---	---	---	---	----	----

2	1	3	8	9	12	15
---	---	---	---	---	----	----

2	1	3	8	9	12	15
---	---	---	---	---	----	----



Improving BubbleSort

BubbleSortImporvedFurther(A):

$n := A.length$

repeat

$lastSwapIdx := -1$

for $j := 1$ **to** $n - 1$

if $A[j] > A[j+1]$

$Swap(A[j], A[j+1])$

$lastSwapIdx := j + 1$

$n := lastSwapIdx - 1$

until $n \leq 1$

$n = 2$

2	1	3	8	9	12	15
---	---	---	---	---	----	----

1	2	3	8	9	12	15
---	---	---	---	---	----	----

1	2	3	8	9	12	15
---	---	---	---	---	----	----

$lastSwapIdx = 2$

$n = 1 \rightarrow$ break loop



Comparison of simple sorting algorithms

- Insertion

- $n(n - 1)/2$ **swaps**, and $n \cdot (n - 1)/2$ **comparisons** -> worst
- $n(n - 1)/4$ **swaps**, and $n \cdot (n - 1)/4$ **comparisons** -> on average

- Selection

- $n - 1$ **swaps**, and $n \cdot (n - 1)/2$ **comparisons**

- Bubble

- $n \cdot (n - 1)/2$ **swaps**, and $n \cdot (n - 1)/2$ **comparisons**

Recall the insertion sort....

Insertion-Sort(A):

```
for  $i := 2$  to  $A.length$ 
     $key := A[i]$ 
     $j := i - 1$ 
    while  $j > 0$  and  $A[j] > key$ 
         $A[j + 1] := A[j]$ 
         $j := j - 1$ 
     $A[j + 1] := key$ 
return  $A$ 
```



Improving Insertion Sorting

- Insertion sorting is effective when:
 - Input size is small
 - The input array is nearly sorted (resulting in few comparisons and swaps)
- Insertion sorting is ineffective when:
 - Elements must move far in array



Improving Insertion Sorting

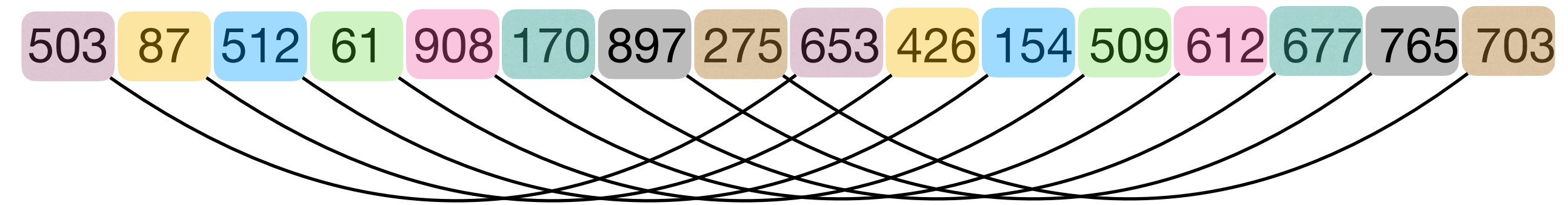
- Allow elements to move large steps
- Bring elements close to final location
 - Make array almost sorted
- Idea: for some decreasing step size h , e.g. $(\dots, 8, 4, 2, 1)$, the sequence must end with 1 (to ensure the correctness of sorting)
 - For each step, sort the array so elements separated by exactly h elements apart are in order.



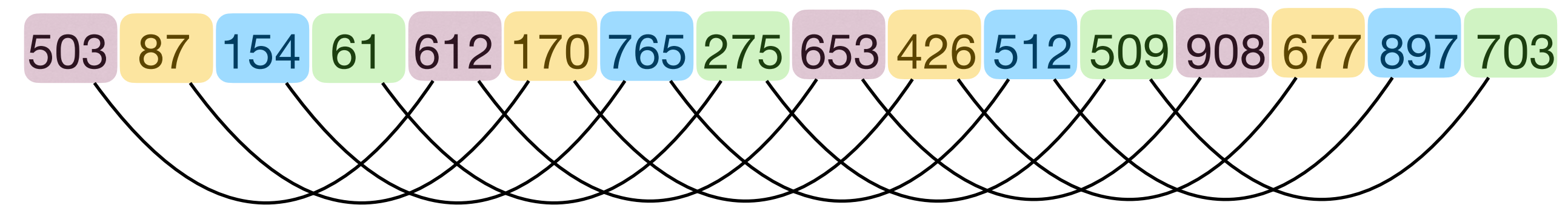
*Shell's method for sorting

Let's first see an example of ShellSort: sort 16 integers.

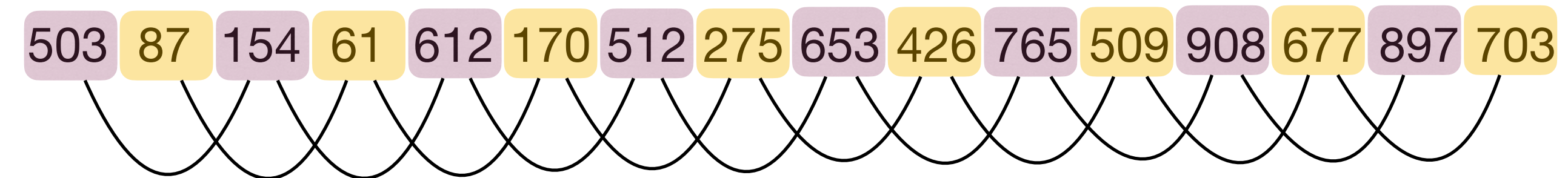
[Pass 1] Group elements of distance **8** together, end up with eight groups each of size two. Sort these groups individually.



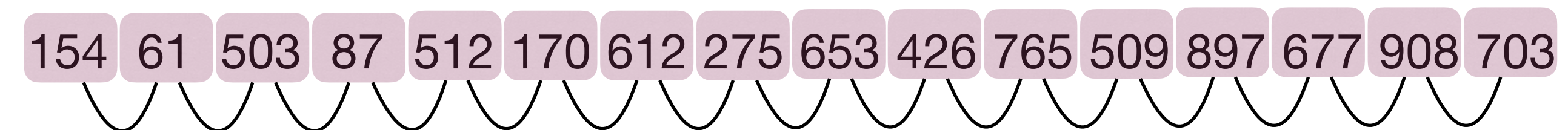
[Pass 2] Group elements of distance **4** together, end up with four groups each of size four. Sort these groups individually.



[Pass 3] Group elements of distance **2** together, end up with two groups each of size eight. Sort these groups individually.



[Pass 4] Group elements of distance **1**, this is just an ordinary sort on all elements.



61 87 154 170 275 426 503 509 512 612 653 677 703 765 897 908



General framework of ShellSort

- To sort n items, define a set of **decreasing distances** $\{d_1, d_2, \dots, d_k\}$ with $d_1 < n$ and $d_k = 1$.
- ShellSort then go through k passes, for the i^{th} pass:
 - ▶ Divide items into d_i groups each of size about n/d_i , and the j^{th} group contains items with index $j, j + d_i, j + 2d_i, j + 3d_i, \dots$
 - ▶ For each of the d_i groups, sort the items in that group. (uses InsertionSort.)



Donald L. Shell



Benefit of ShellSort

- In a sequence of items $\langle a_1, a_2, \dots, a_n \rangle$, if $i < j$ and $a_i > a_j$, then the pair (a_i, a_j) is call an **inversion**.
- The process of sorting is to correct all inversions!
- Earlier passes in `ShellSort` reduce number of inversions, making the sequence “closer” to being sorted.
- `InsertionSort` performs better (i.e., faster) as the input sequence becomes “closer” to being sorted.



Ideal versus Reality

- Unfortunately, `ShellSort` is not that fast, at least when using Shell's original distances...
- Upper bound on the runtime of `ShellSort`:
 - Assume we have n items where n is some power of two.
 - The distances are $n/2, n/4, \dots, 1$.
 - For the i^{th} pass, we run $n/2^i$ instances of `InsertionSort`, each having to sort 2^i items.

- So the total runtime is
$$\sum_{i=1}^{(\lg n)-1} (n/2^i \cdot O(2^{2i})) = O(n^2)$$

- Will `ShellSort` actually perform so poor?



ShellSort can be slow!

- When using Shell's original distances, the runtime of `ShellSort` can be $\Theta(n^2)$ for certain input sequences.
- Example: input is $[n]$, where $[n/2]$ are in even positions, and $[n] \setminus [n/2]$ are in odd positions.

8	0	9	1	10	2	11	3	12	4	13	5	14	6	15	7
---	---	---	---	----	---	----	---	----	---	----	---	----	---	----	---

- Then, before the last pass, no pair (a_i, a_j) where i and j are of different parity is ever compared!
- In the last pass, $\Theta(n^2)$ work has to be done!



Choice of distances matters, a lot!

General term ($k \geq 1$)	Concrete gaps	Worst-case time complexity	Author and year of publication
$\left\lfloor \frac{N}{2^k} \right\rfloor$	$1, 2, \dots, \left\lfloor \frac{N}{4} \right\rfloor, \left\lfloor \frac{N}{2} \right\rfloor$	$\Theta(N^2)$ [e.g. when $N = 2^p$]	Shell, 1959 ^[4]
$2 \left\lfloor \frac{N}{2^{k+1}} \right\rfloor + 1$	$1, 3, \dots, 2 \left\lfloor \frac{N}{8} \right\rfloor + 1, 2 \left\lfloor \frac{N}{4} \right\rfloor + 1$	$\Theta\left(N^{\frac{3}{2}}\right)$	Frank & Lazarus, 1960 ^[8]
$2^k - 1$	$1, 3, 7, 15, 31, 63, \dots$	$\Theta\left(N^{\frac{3}{2}}\right)$	Hibbard, 1963 ^[9]
$2^k + 1$, prefixed with 1	$1, 3, 5, 9, 17, 33, 65, \dots$	$\Theta\left(N^{\frac{3}{2}}\right)$	Papernov & Stasevich, 1965 ^[10]
Successive numbers of the form $2^p 3^q$ (3-smooth numbers)	$1, 2, 3, 4, 6, 8, 9, 12, \dots$	$\Theta(N \log^2 N)$	Pratt, 1971 ^[1]
$\frac{3^k - 1}{2}$, not greater than $\left\lceil \frac{N}{3} \right\rceil$	$1, 4, 13, 40, 121, \dots$	$\Theta\left(N^{\frac{3}{2}}\right)$	Knuth, 1973, ^[3] based on Pratt, 1971 ^[1]
$\prod_I a_q$, where $a_0 = 3$ $a_q = \min \left\{ n \in \mathbb{N} : n \geq \left(\frac{5}{2}\right)^{q+1}, \forall p: 0 \leq p < q \Rightarrow \gcd(a_p, n) = 1 \right\}$ $I = \left\{ 0 \leq q < r \mid q \neq \frac{1}{2}(r^2 + r) - k \right\}$ $r = \left\lfloor \sqrt{2k + \sqrt{2k}} \right\rfloor$	$1, 3, 7, 21, 48, 112, \dots$	$O\left(N^{1 + \sqrt{\frac{8 \ln(5/2)}{\ln(N)}}}\right)$	Incerpi & Sedgewick, 1985, ^[11] Knuth ^[3]



Quick Sort





A unified view of many sorting algorithms

Divide problem into subproblems. Conquer subproblems recursively. Combine solutions of subproblems.

- Divide the input into size 1 and size $n - 1$.
 - InsertionSort, easy to divide, combine needs efforts.
 - SelectionSort, divide needs efforts, easy to combine.
- Divide the input into two parts each of same size.
 - MergeSort, easy to divide, combine needs efforts.
- Divide the input into two parts of **approximately** same size.
 - QuickSort, divide needs efforts, easy to combine.



The QuickSort Algorithm

- **Basic idea:**

- ▶ Given an array A of n items.
 - Choose one item x in A as the **pivot**.
 - Use the pivot to **partition** the input into B and C , so that items in B are $\leq x$, and items in C are $> x$.
 - Recursively sort B and C .
 - Output $\langle B, x, C \rangle$.

QuickSortAbs(A):

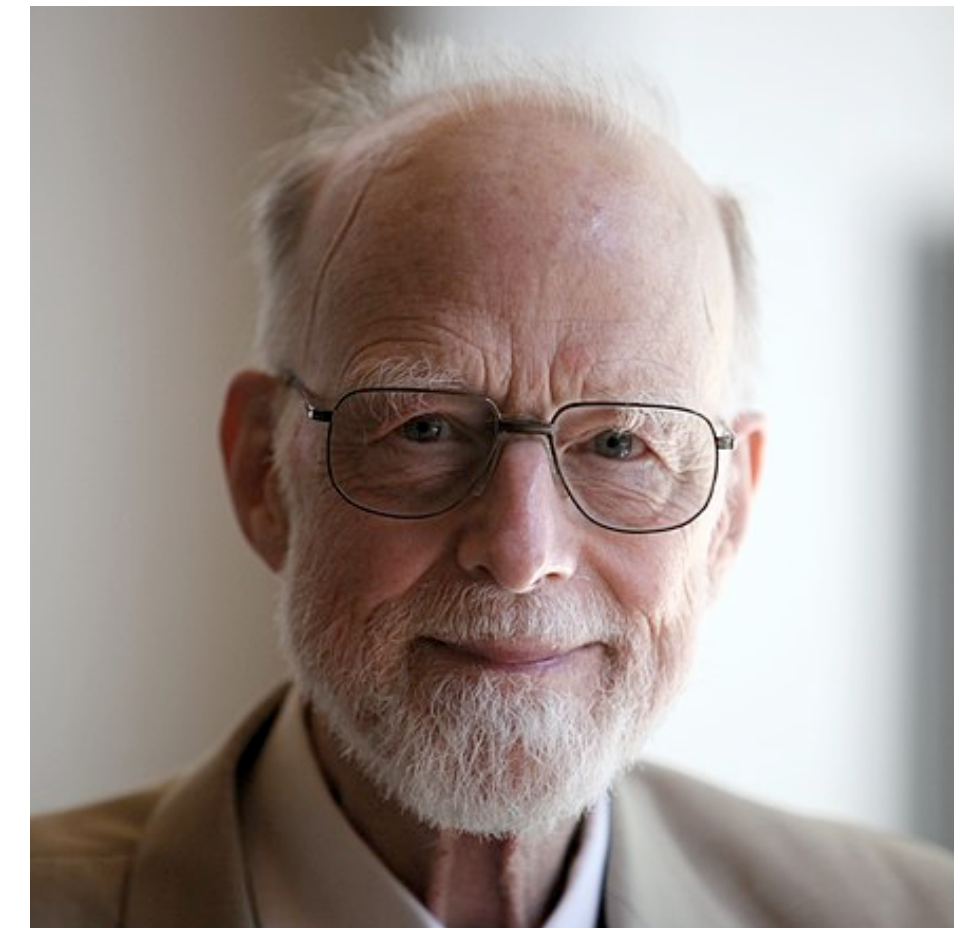
$x := \text{GetPivot}(A)$

$\langle B, C \rangle := \text{Partition}(A, x)$

$\text{QuickSortAbs}(B)$

$\text{QuickSortAbs}(C)$

return $\text{Concatenate}(B, x, C)$



Tony Hoare



Choosing the pivot

- Ideally the pivot should partition the input into two parts of roughly the same size (we'll see why later).
 - Select the “middle” element, the “first” element, or the “last” element?
 - Or using “Median-of-three” technique, e.g., $A[1]$, $A[n]$, $A[n/2]$, median of $\{ A[1], A[n], A[n/2] \}$?
- For every **simple deterministic** method of choosing pivot, we can construct corresponding “**bad input**”.
- For now just use the **last** item as the pivot.



The Partition Procedure

- Allocate array B of size n .
- Sequentially go through $A[1 \dots (n-1)]$, put small items at the left side of B , and large items at the right side of B .
- Finally put the pivot in the (only) remaining position.
- $\Theta(n)$ time, $\Theta(n)$ space, unstable.
- Can we do better, and how?

Partition(A):

$x := A[n], l := 1, r := n$

for $i := 1$ **to** $n - 1$

if $A[i] \leq x$

$B[l] := A[i]$

$l++$

else

$B[r] := A[i]$

$r--$

$B[l] := x$

return $\langle B, l \rangle$



In-place Partition Procedure

- Basic idea: sequentially go through A , use **swap** operations to move small items to the left part of A ; thus the right part of A naturally contains large items.

InplacePartition(A, p, r):

$i := p - 1$

for $j := p$ **to** $r - 1$

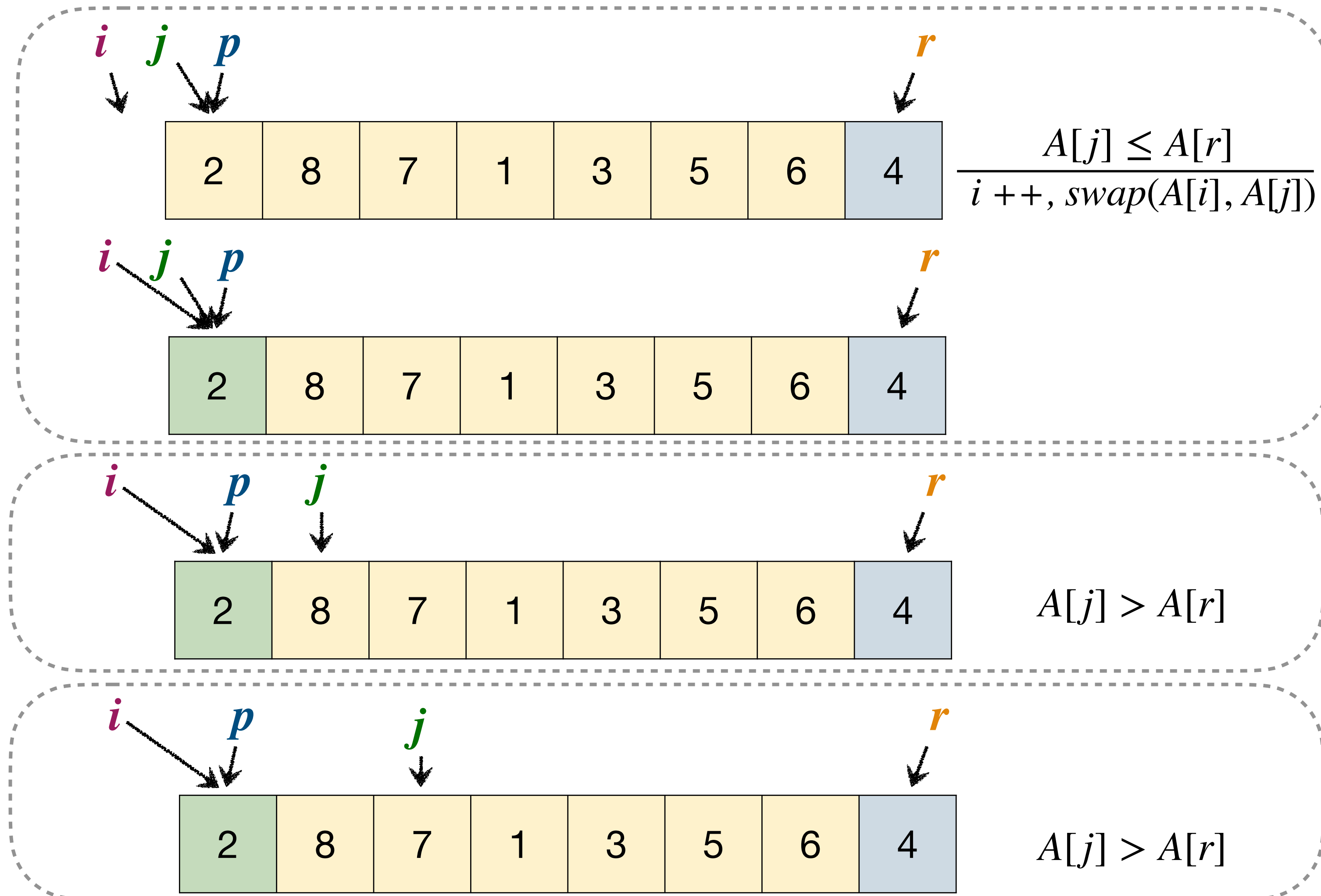
if $A[j] \leq A[r]$

$i := i + 1$

$\text{Swap}(A[i], A[j])$

$\text{Swap}(A[i+1], A[r])$

return $i + 1$





In-place Partition Procedure

- Basic idea: sequentially go through A , use **swap** operations to move small items to the left part of A ; thus the right part of A naturally contains large items.

InplacePartition(A, p, r):

$i := p - 1$

for $j := p$ **to** $r - 1$

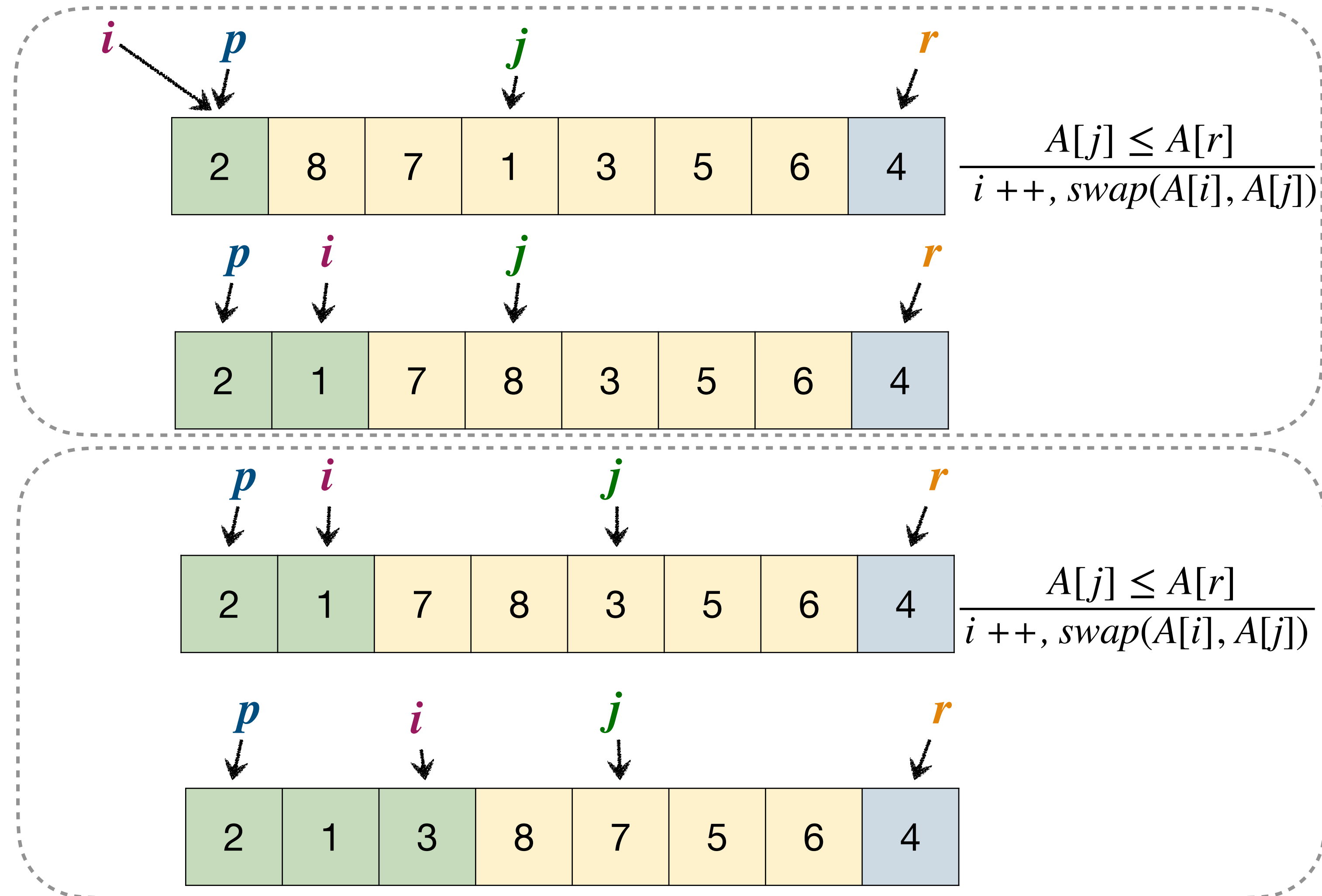
if $A[j] \leq A[r]$

$i := i + 1$

$\text{Swap}(A[i], A[j])$

$\text{Swap}(A[i+1], A[r])$

return $i + 1$





In-place Partition Procedure

- Basic idea: sequentially go through A , use **swap** operations to move small items to the left part of A ; thus the right part of A naturally contains large items.

InplacePartition(A, p, r):

$i := p - 1$

for $j := p$ **to** $r - 1$

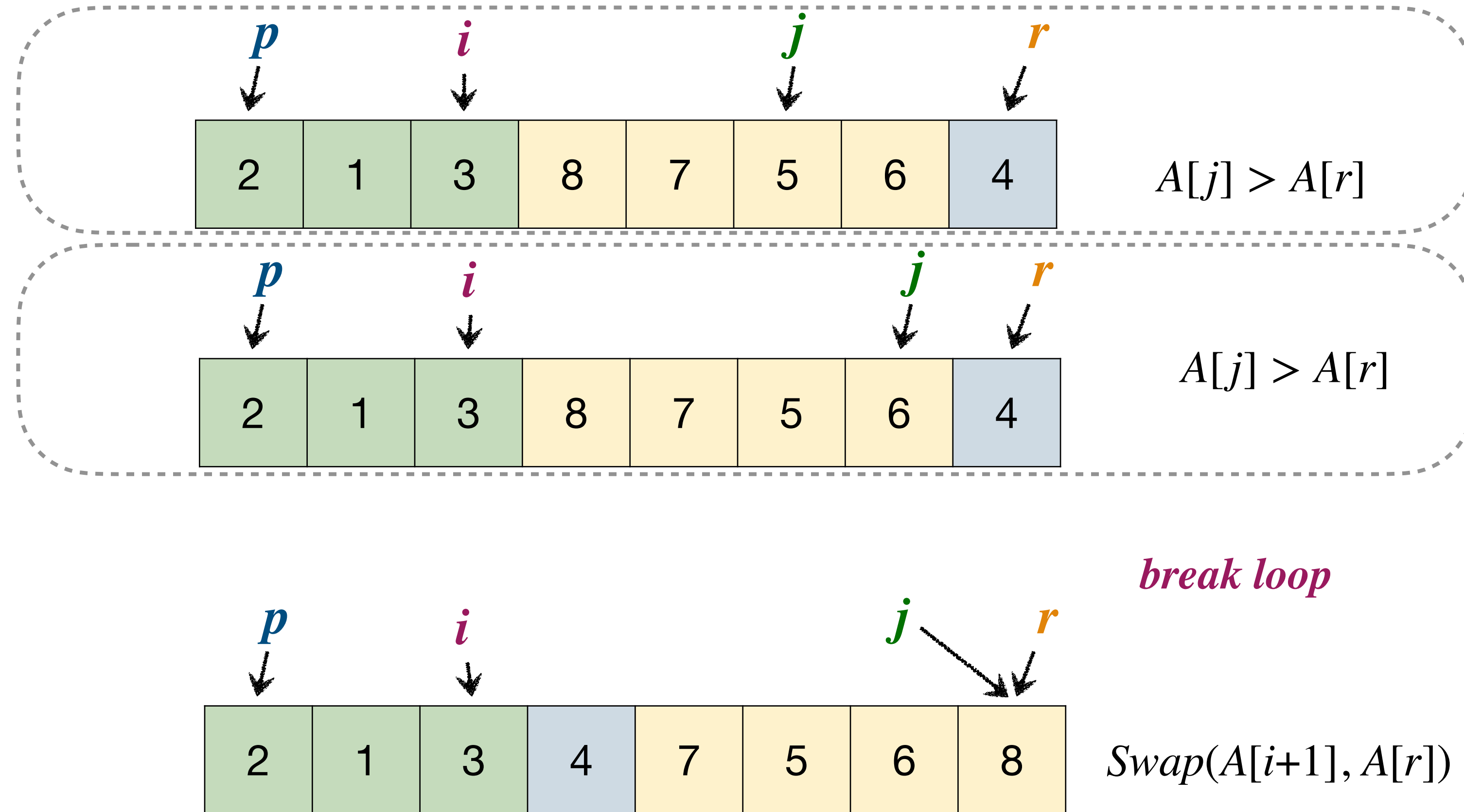
if $A[j] \leq A[r]$

$i := i + 1$

$\text{Swap}(A[i], A[j])$

$\text{Swap}(A[i+1], A[r])$

return $i + 1$



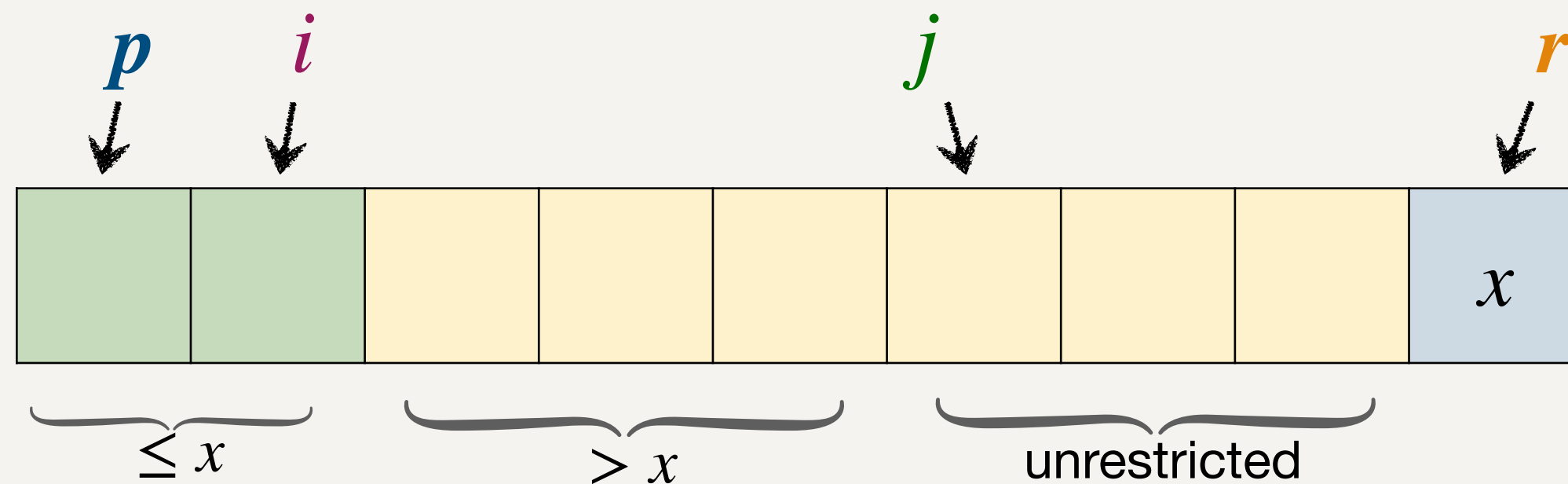


Analysis of In-place Partition Procedure

Correctness

- **Claim:** at the beginning of any iteration, for any index k :

- If $k \in [p, i]$, then $A[k] \leq A[r]$;
- If $k \in [i + 1, j - 1]$, then $A[k] > x$;
- If $k = r$, then $A[k] = A[r]$.



InplacePartition(A, p, r):

$i := p - 1$

for $j := p$ **to** $r - 1$

if $A[j] \leq A[r]$

$i := i + 1$

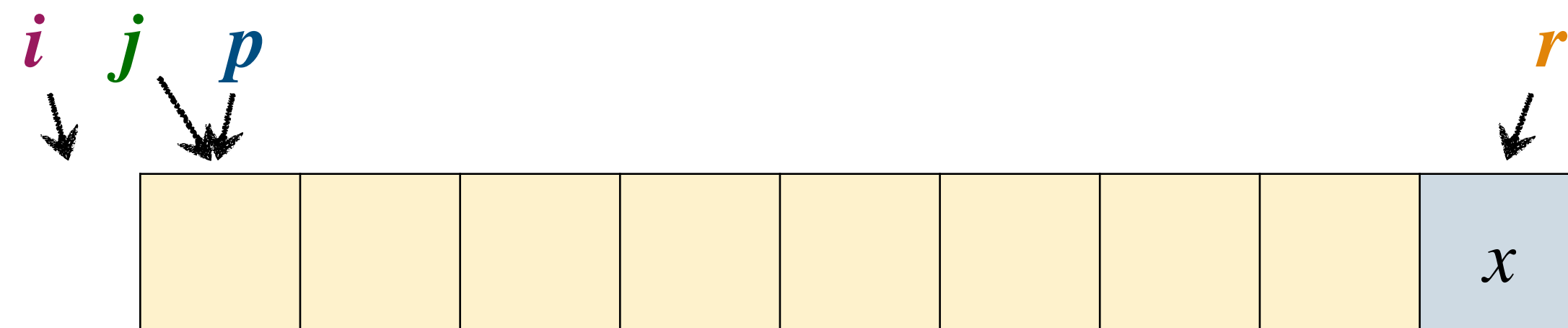
$\text{Swap}(A[i], A[j])$

$\text{Swap}(A[i+1], A[r])$

return $i + 1$

- **Proof:** we use induction.

- **[Basis]** Trivially holds.





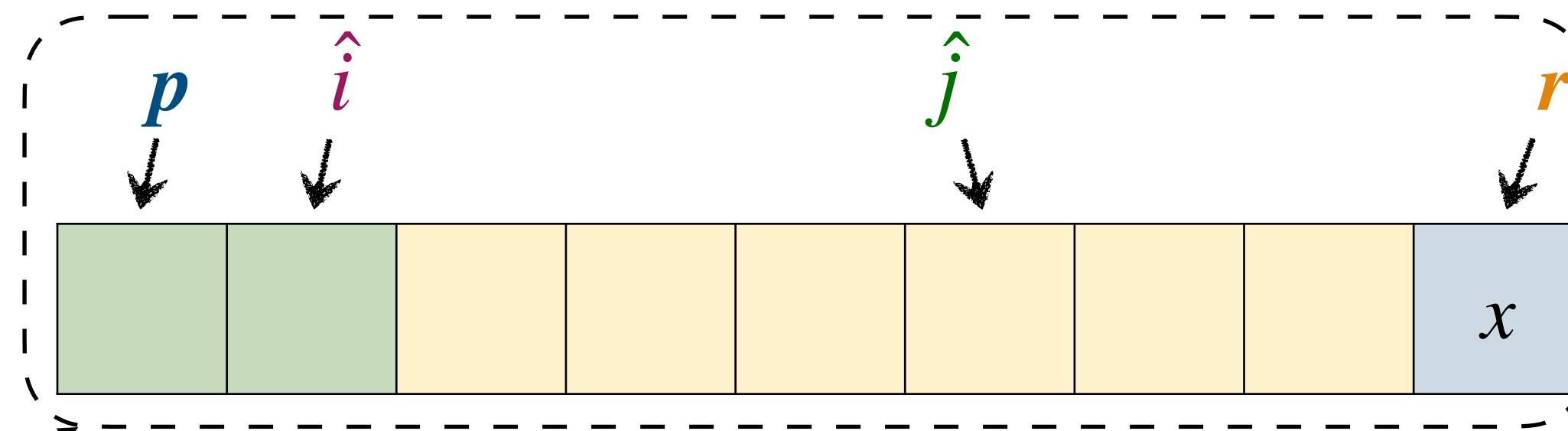
Analysis of In-place Partition Procedure

Correctness

- **Proof:** we use induction.

‣ **[Basis]** Trivially holds.

‣ **[Inductive step]** Assume at the beginning of some iteration we have $i = \hat{i}$ and $j = \hat{j}$, and the stated properties hold. In this iteration:



InplacePartition(A, p, r):

$i := p - 1$

for $j := p$ **to** $r - 1$

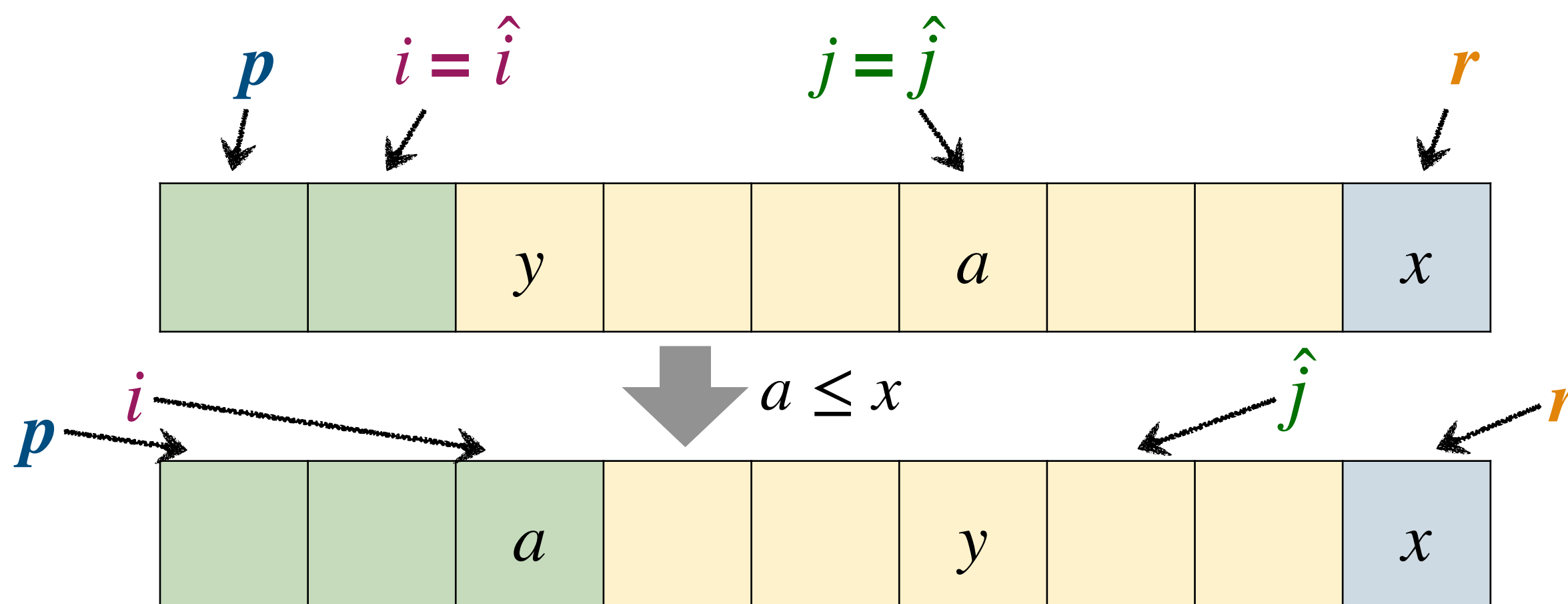
if $A[j] \leq A[r]$

$i := i + 1$

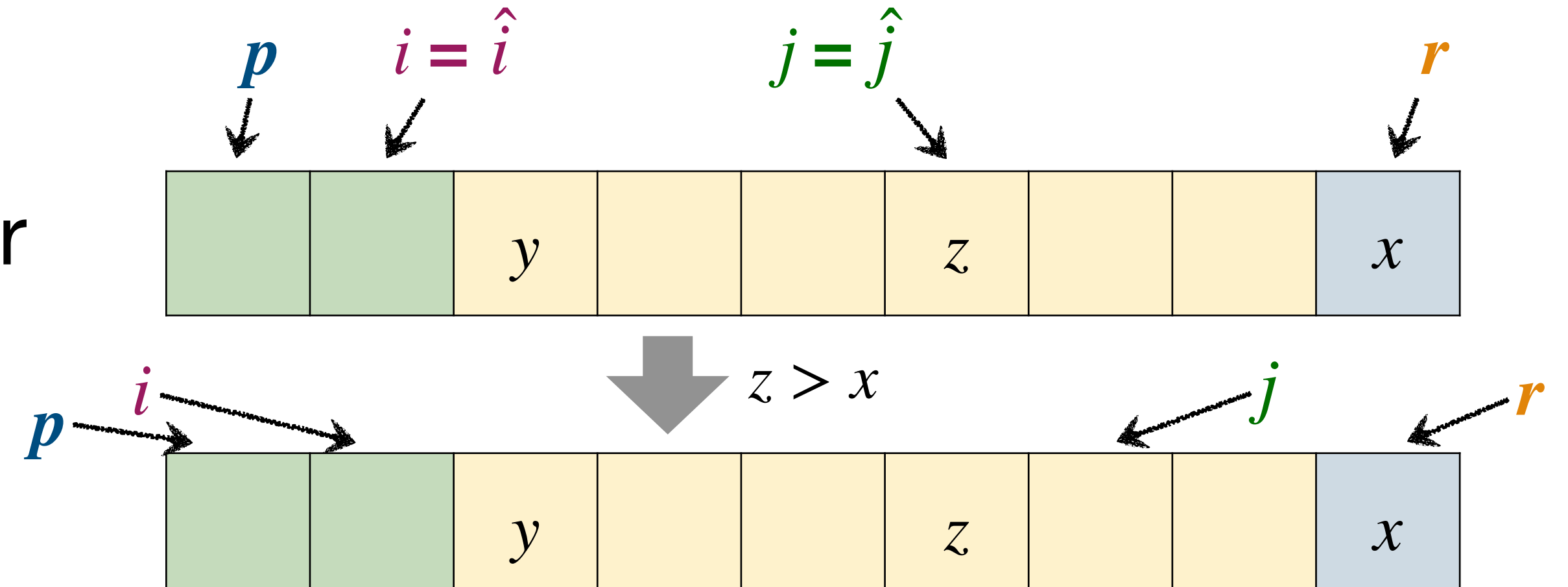
$Swap(A[i], A[j])$

$Swap(A[i+1], A[r])$

return $i + 1$



or





Analysis of In-place Partition Procedure

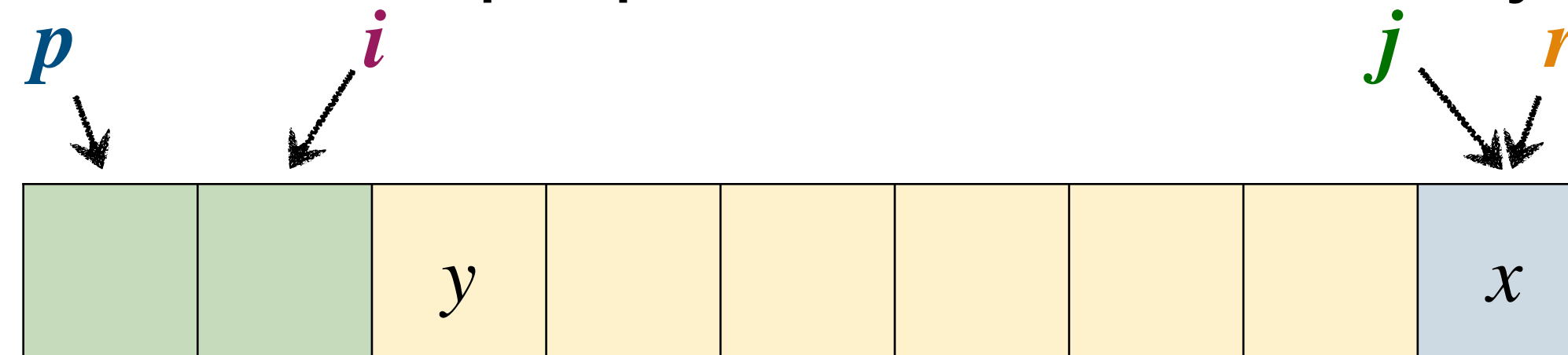
Correctness

- **Proof:** we use induction.

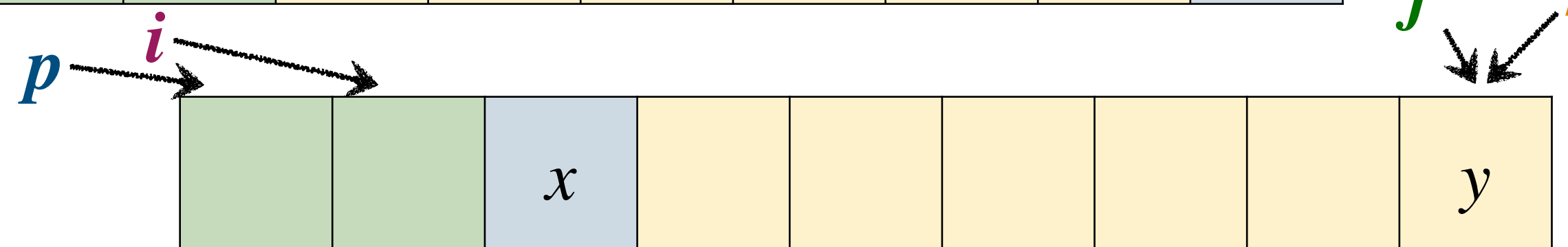
- ▶ **[Basis]** Trivially holds.

- ▶ **[Inductive step]** Assume at the beginning of some iteration we have $i = \hat{i}$ and $j = \hat{j}$, and the stated properties hold. Then they hold after this iteration.

- eventually, when $j = r$:



- Swap $A[i + 1]$ and $A[r]$



InplacePartition(A, p, r):

$i := p - 1$

for $j := p$ **to** $r - 1$

if $A[j] \leq A[r]$

$i := i + 1$

 Swap($A[i], A[j]$)

Swap($A[i+1], A[r]$)

return $i + 1$

During execution, we only *swap* items, no *addition /deletion*.

So *InplacePartition* correctly partitions the input array.



The QuickSort Algorithm

InplacePartition(A, p, r):

$i := p - 1$

for $j := p$ **to** $r - 1$

if $A[j] \leq A[r]$

$i := i + 1$

$Swap(A[i], A[j])$

$Swap(A[i+1], A[r])$

return $i + 1$

QuickSort(A, p, r):

if $p < r$

$q := InplacePartition(A, p, r)$

$QuickSort(A, p, q - 1)$

$QuickSort(A, q + 1, r)$

- Performance of InplacePartition:

- $\Theta(|r - p|)$ time (i.e., linear time);
- $O(1)$ space; unstable.

- Performance of QuickSort?

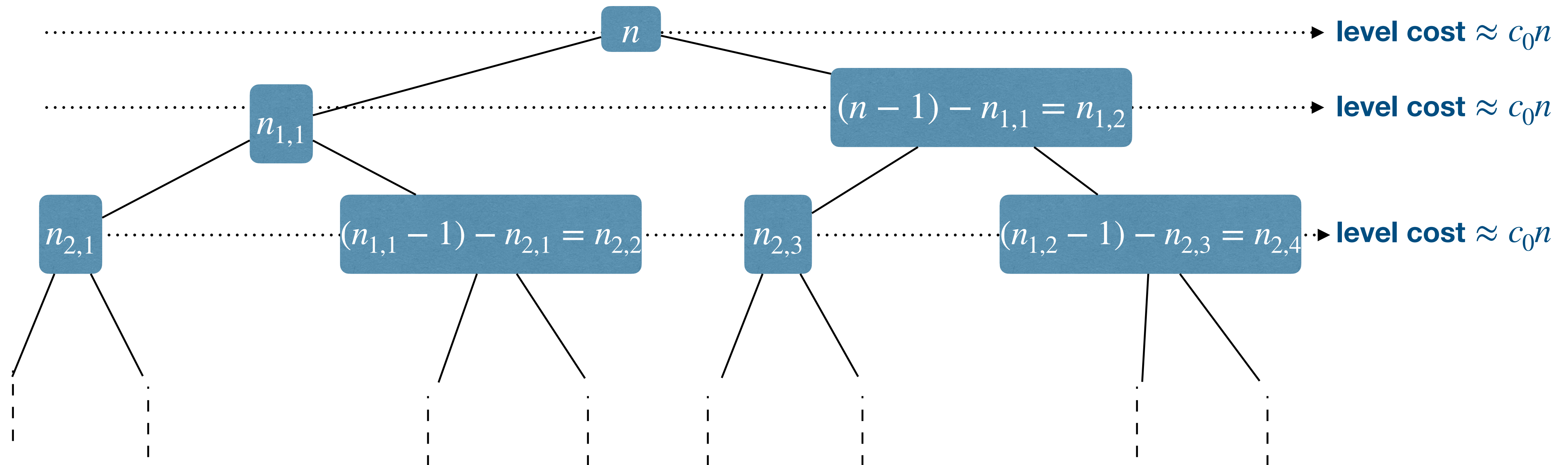
Note: Although quicksort sorts in-place, the amount of memory it use aside from the array being sorted is not constant.

Since each recursive call requires additional amount of space on the runtime stack. How many of them?



Performance of QuickSort

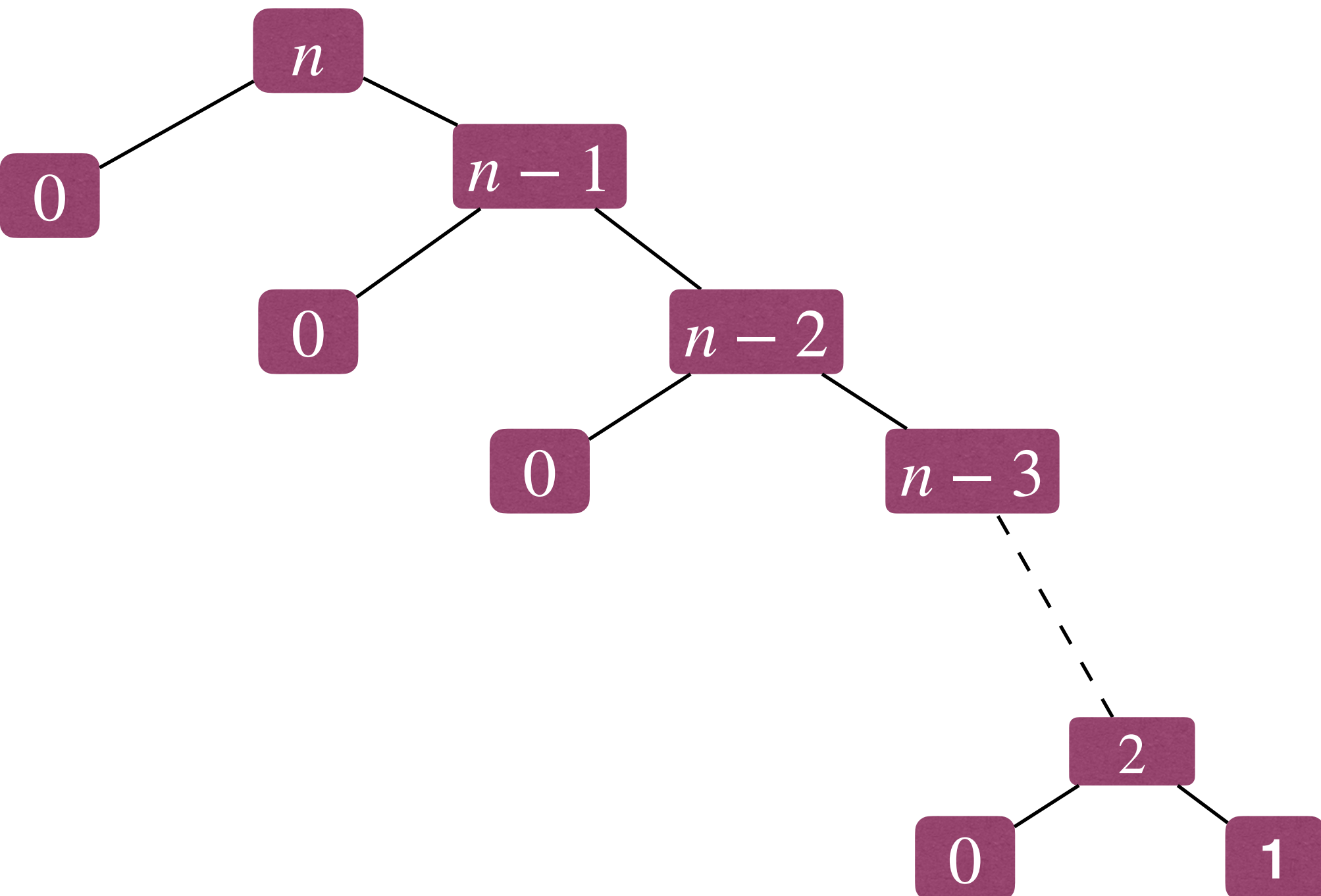
- Cost at each level is: $c_0(n - m)$, where m is number of pivots removed in lower level Partition.
 - If the partition is “balanced”, then there will be few levels.
 - If the partition is “balanced”, then m will increase rapidly.



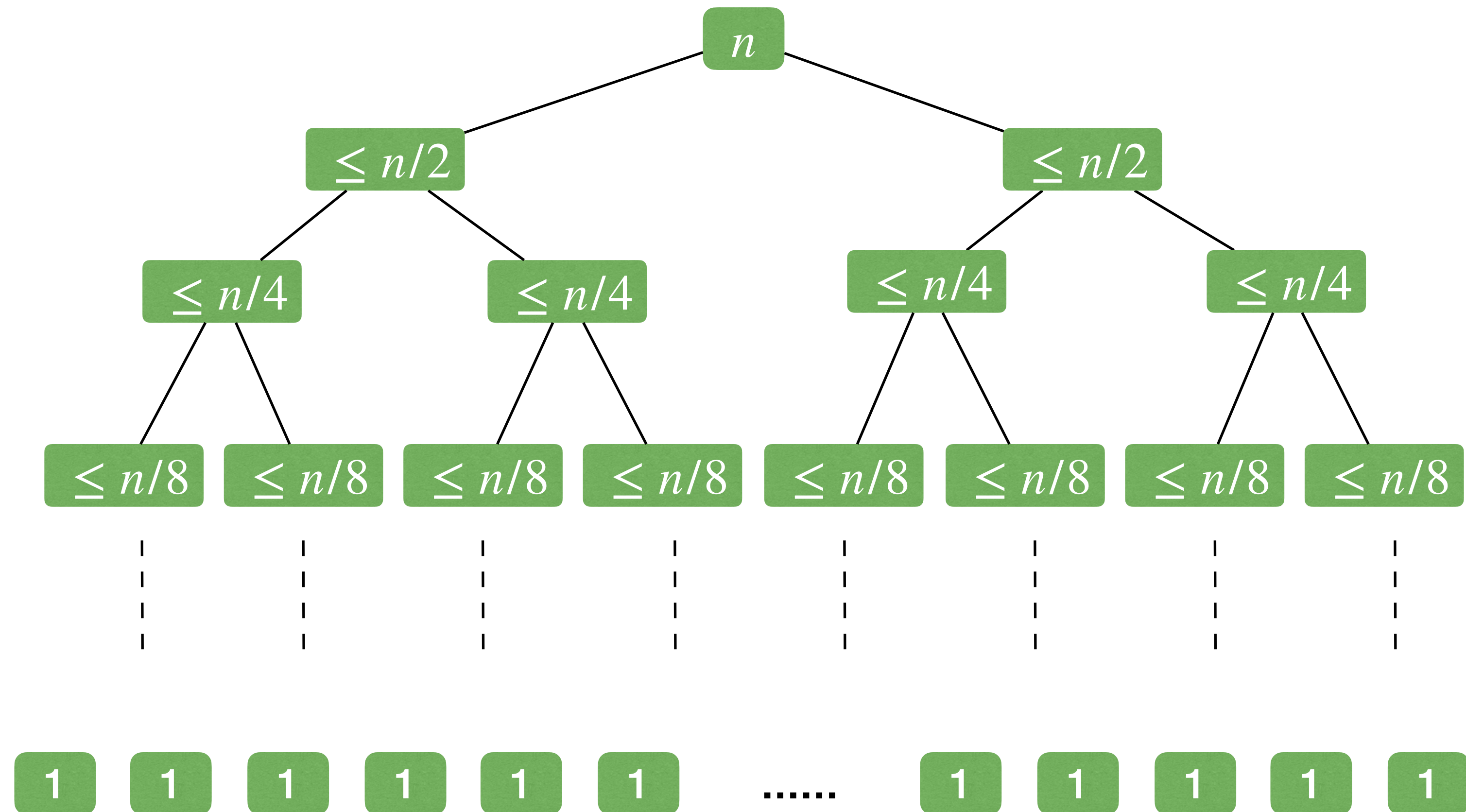


Performance of QuickSort

Worst case



Best case





Performance of QuickSort

- Recurrence for the worse-case runtime of `QuickSort`:

- $T(n) = \max_{0 \leq q \leq n-1} (T(q) + T(n - q - 1)) + c_0 n$

- Guess $T_n \leq cn^2$, and we now verify:

- $T(n) \leq \max_{0 \leq q \leq n-1} (cq^2 + c(n - q - 1)^2) + c_0 n$

$$= c \cdot \max_{0 \leq q \leq n-1} (q^2 + (n - q - 1)^2) + c_0 n$$

when $q = 0$ or $q = n - 1$

$$\leq c(n - 1)^2 + c_0 n = cn^2 - c(2n - 1) + c_0 n$$

$$\leq cn^2 \quad \rightarrow T(n) = O(n^2)$$

QuickSort(A, p, r):

if $p < r$

$q := \text{InplacePartition}(A, p, r)$

$\text{QuickSort}(A, p, q - 1)$

$\text{QuickSort}(A, q + 1, r)$



Performance of QuickSort

- “Balanced” partition gives best case performance.

- $T(n) \leq T(n/2) + T(n/2) + \Theta(n)$ implies
 $T(n) = O(n \log n)$.

- Partition does not need to be perfectly balanced, we only need each split to be **constant** proportionality.

- $T(n) \leq T(dn) + T((1 - d)n) + \Theta(n)$ where
 $d = \Theta(1)$.

QuickSort(A, p, r):

if $p < r$

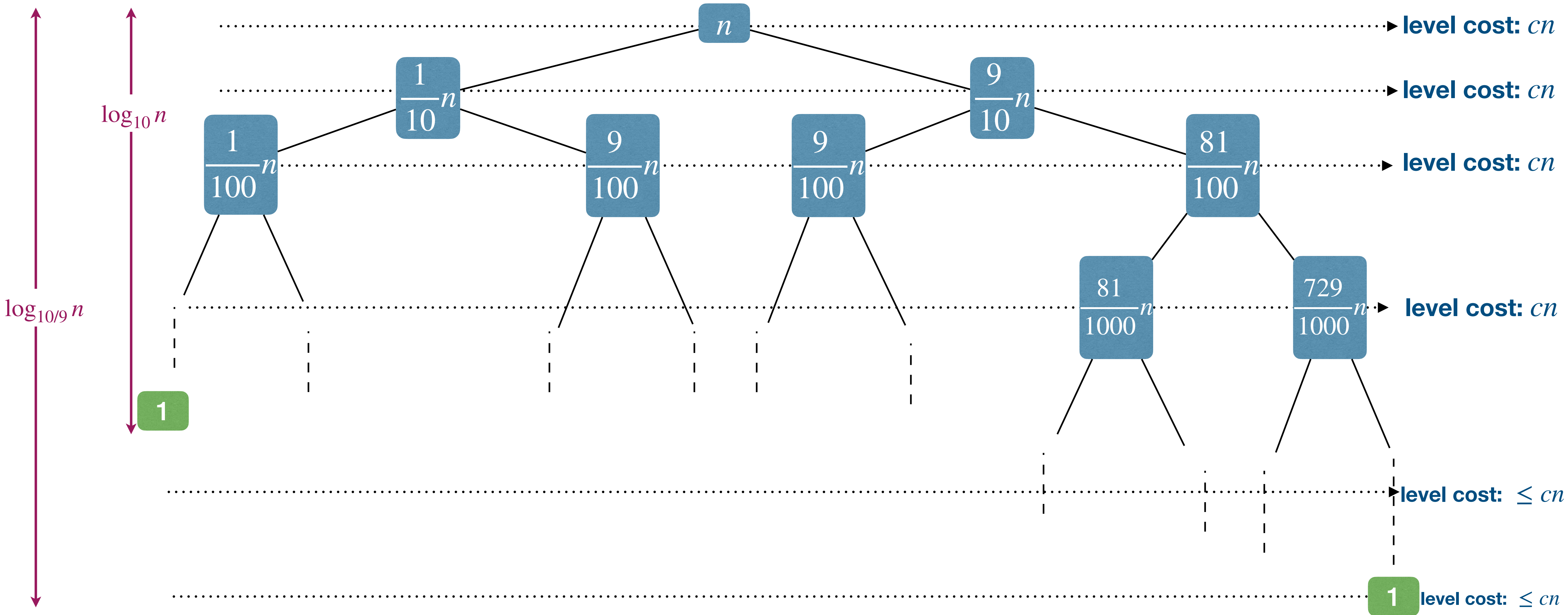
$q := \text{InplacePartition}(A, p, r)$

$\text{QuickSort}(A, p, q - 1)$

$\text{QuickSort}(A, q + 1, r)$



Performance of QuickSort



$O(n \log n)$



Performance of QuickSort

- The performance of the best is $\Theta(n \log n)$, while the worst is $\Theta(n^2)$
 - What about the performance in general?
- **Average-case analysis:** the **expected** time of algorithm over all inputs of size n (i.e., $\mathcal{X}_n$) :
$$A(n) = \sum_{x \in \mathcal{X}_n} T(x) \cdot Pr(x)$$
 - In order to perform a probabilistic analysis, we must use knowledge of, or make assumptions about, the distribution of (something about) the inputs.



Performance of QuickSort

- For `QuickSort`, particular values in the array are not important, instead, the **relative ordering** of the values is what matters (since `QuickSort` is comparison-based).
- Therefore, it is important to focus on the permutation of input numbers. A readable assumption is that all permutations of the input numbers are equally likely.
 - To make the analysis simple, we also assume that the elements are distinct (duplicate values will be discussed later).



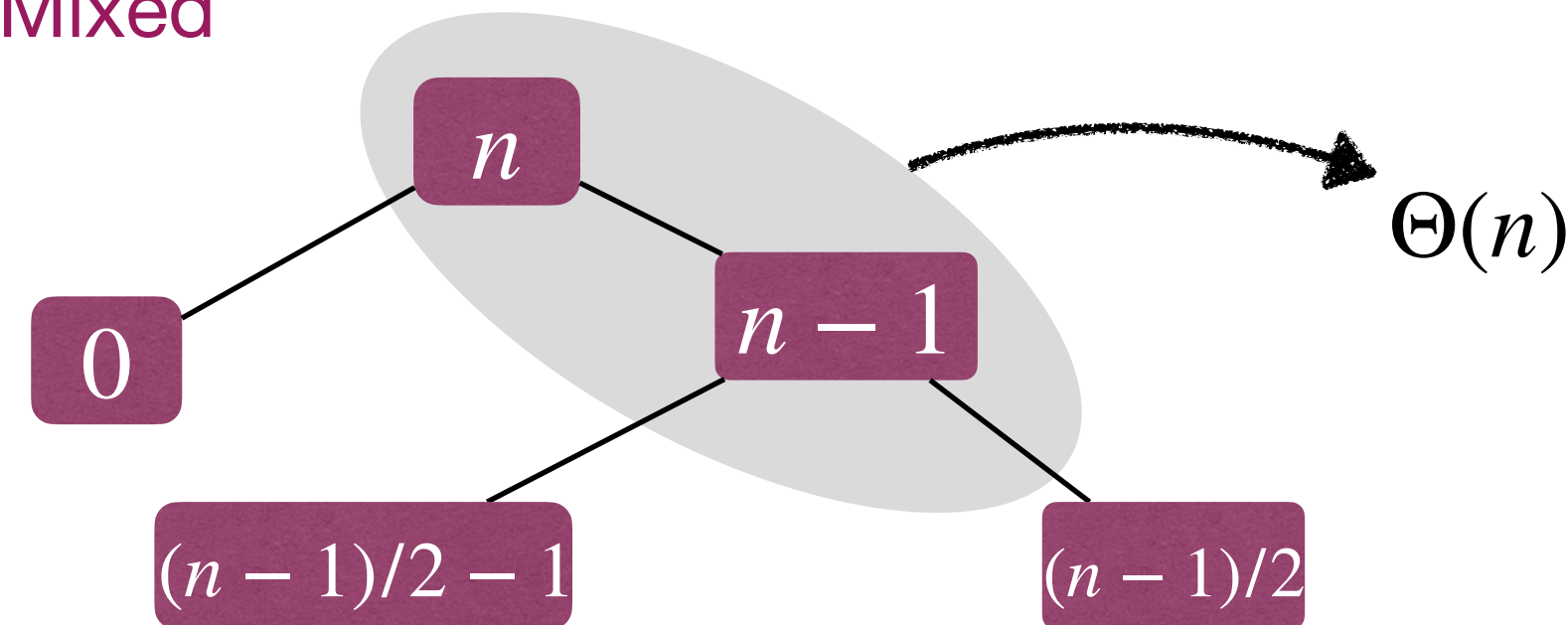
Performance of QuickSort

- Before making rigorous analysis, we can first gain some intuition about the average performance.
 - ▶ When `QuickSort` runs on a random input array, we expect that some of the splits will be reasonably **well balanced** and that some will be **fairly unbalanced**.
 - ▶ In the average case, `Partition` produces a **mix** of “**good**” and “**bad**” splits. That is, in a recursion tree, the good and bad splits are distributed randomly throughout the tree.

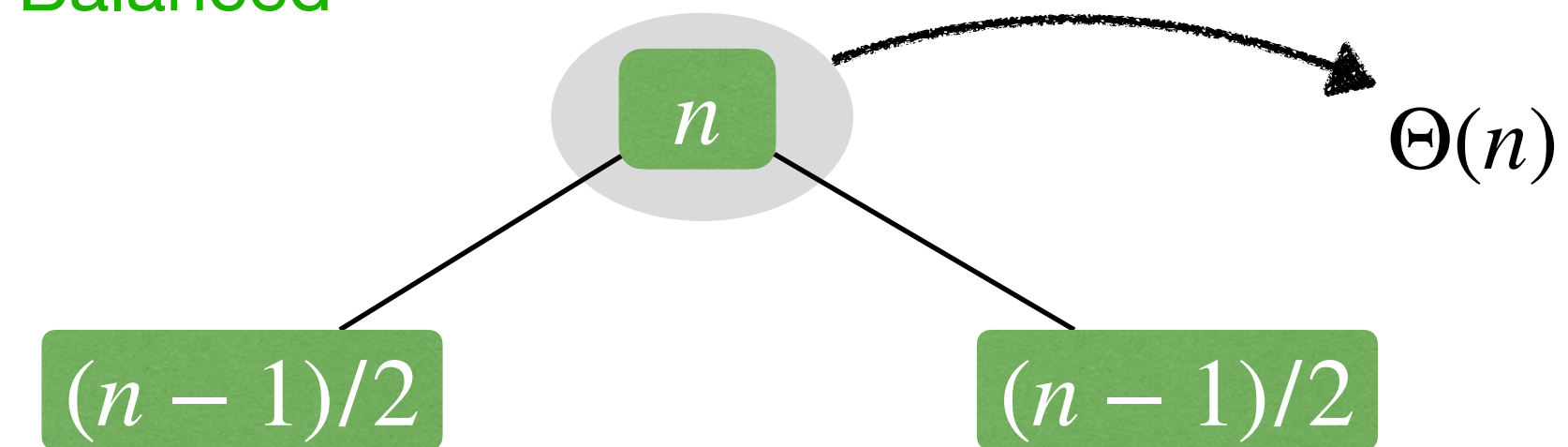


Performance of QuickSort

Mixed



Balanced



- Further, for the sake of intuition, suppose that the good and bad splits **alternate** levels in the tree, and that the good splits are best case splits and the bad splits are worst-case splits.
 - As an example in the above, the “mixed” Partition produces two “ $(n-1)/2$ ” subarrays at the cost of $\Theta(n) + \Theta(n-1) = \Theta(n)$, while the “balanced” Partition does so at the cost of $\Theta(n)$.
 - The cost of “bad” Partition can be absorbed by recent “good” Partition, without affecting time complexity asymptotically $\rightarrow$ “mixed” Partition is at most constant factor worse than “balanced” Partition.
 - Therefore, the **average runtime** of QuickSort is $O(n \log n)$ (rigorously proved later).



Randomized QuickSort

- Picking “good” pivot is important for the performance? but how do we do it?
 - On choosing pivot: first, middle, last, median of three, ...?
- Any simple **deterministic** mechanism could fail! (If the input is given by an “adversary” that knows the algorithm.)

- Choose pivot (uniformly) **at random!**
 - Since the choice is randomly made, there is a good chance (constant probability) that we choose a “good” pivot.
- The above claim holds even if the input is given by an “adversary” that knows the algorithm (but not the random bits the algorithm uses).

RandQuickSort(A, p, r):

if $p < r$

$i := \text{Random}(p, r)$

$\text{Swap}(A[r], A[i])$

$q := \text{InplacePartition}(A, p, r)$

$\text{RandQuickSort}(A, p, q - 1)$

$\text{RandQuickSort}(A, q + 1, r)$



Randomized QuickSort

RandQuickSort(A, p, r):

if $p < r$

$i := \text{Random}(p, r)$
 $\text{Swap}(A[r], A[i])$

$q := \text{InplacePartition}(A, p, r)$

$\text{RandQuickSort}(A, p, q - 1)$

$\text{RandQuickSort}(A, q + 1, r)$

Constant time

Two calls

InplacePartition(A, p, r):

$i := p - 1$

for $j := p$ **to** $r - 1$

if $A[j] \leq A[r]$

$i := i + 1$

$\text{Swap}(A[i], A[j])$

$\text{Swap}(A[i+1], A[r])$

return $i + 1$

$O(\text{number of comparisons})$



Randomized QuickSort

- Cost of a call to `RndQuickSort`:
 - Choose a pivot in $\Theta(1)$ time;
 - Run `InplacePartition`, the cost is $O(\text{number of comparisons})$.
 - Need to call `RndQuickSort` twice, the calling process (**not** the subroutines themselves) needs $\Theta(1)$ time.
- Total cost of `RndQuickSort`:
 - Time for choosing pivots $O(n)$, since each node can be pivot at most once!
 - All calls to `InplacePartition`, $O(\text{total number of comparisons})$.
 - Total time for call `RndQuickSort` $O(2n)$, since each time a pivot is chosen, two `RndQuickSort` calls are made.

In an execution of `RndQuickSort`, the cost is $O(n) + O(\text{total number of comparisons})$



Randomized QuickSort

Cost of `RndQuickSort` is $O(n + X)$, where X is a random variable denoting the number of comparisons happened in `InplacePartition` throughout entire execution.

- Each of pair of items is compared at most once! (Items only compare with pivots, and each item can be the pivot at most once.)
- For ease of analysis, we let's index the elements of the array A by their position in the **sorted** output, rather than their **position** in the input.
 - For all the elements, we refer them to be $z_1, z_2, \dots, z_n$, with $z_1 < z_2 < \dots < z_n$.

- Let $X_{ij} = I \{ z_i \text{ is ever compared to } z_j \}$, here I is an indicator random variable $I(H) = \begin{cases} 1 & H \text{ happens} \\ 0 & H \text{ not happen} \end{cases}$

- $$\mathbb{E}[X] = \mathbb{E} \left[\sum_{i=1}^{n-1} \sum_{j=i+1}^n X_{ij} \right] = \sum_{i=1}^{n-1} \sum_{j=i+1}^n \mathbb{E}[X_{ij}] = \sum_{i=1}^{n-1} \sum_{j=i+1}^n \Pr(X_{ij} = 1)$$



Randomized QuickSort

- Let $Z_{ij} = \{z \mid z \in A, z_i \leq z \leq z_j\}$, where $i \leq j$, let $\hat{z}_{ij}$ be the first item in Z_{ij} that is chosen as a pivot. Then z_i and z_j are compared iff $\hat{z}_{ij} = z_i$ or $\hat{z}_{ij} = z_j$. (Items from Z_{ij} stay in same split until some pivot is chosen from Z_{ij}).
- $Pr(X_{ij} = 1) = Pr(\hat{z}_{ij} = z_i) + Pr(\hat{z}_{ij} = z_j) = \frac{2}{j - i + 1}$
- $\mathbb{E}[X] = \sum_{i=1}^{n-1} \sum_{j=i+1}^n \frac{2}{j - i + 1}$, let $k = j - i$, $\mathbb{E}[X] = \sum_{i=1}^{n-1} \sum_{k=1}^{n-i} \frac{2}{k + 1} < \sum_{i=1}^{n-1} \sum_{k=1}^n \frac{2}{k}$



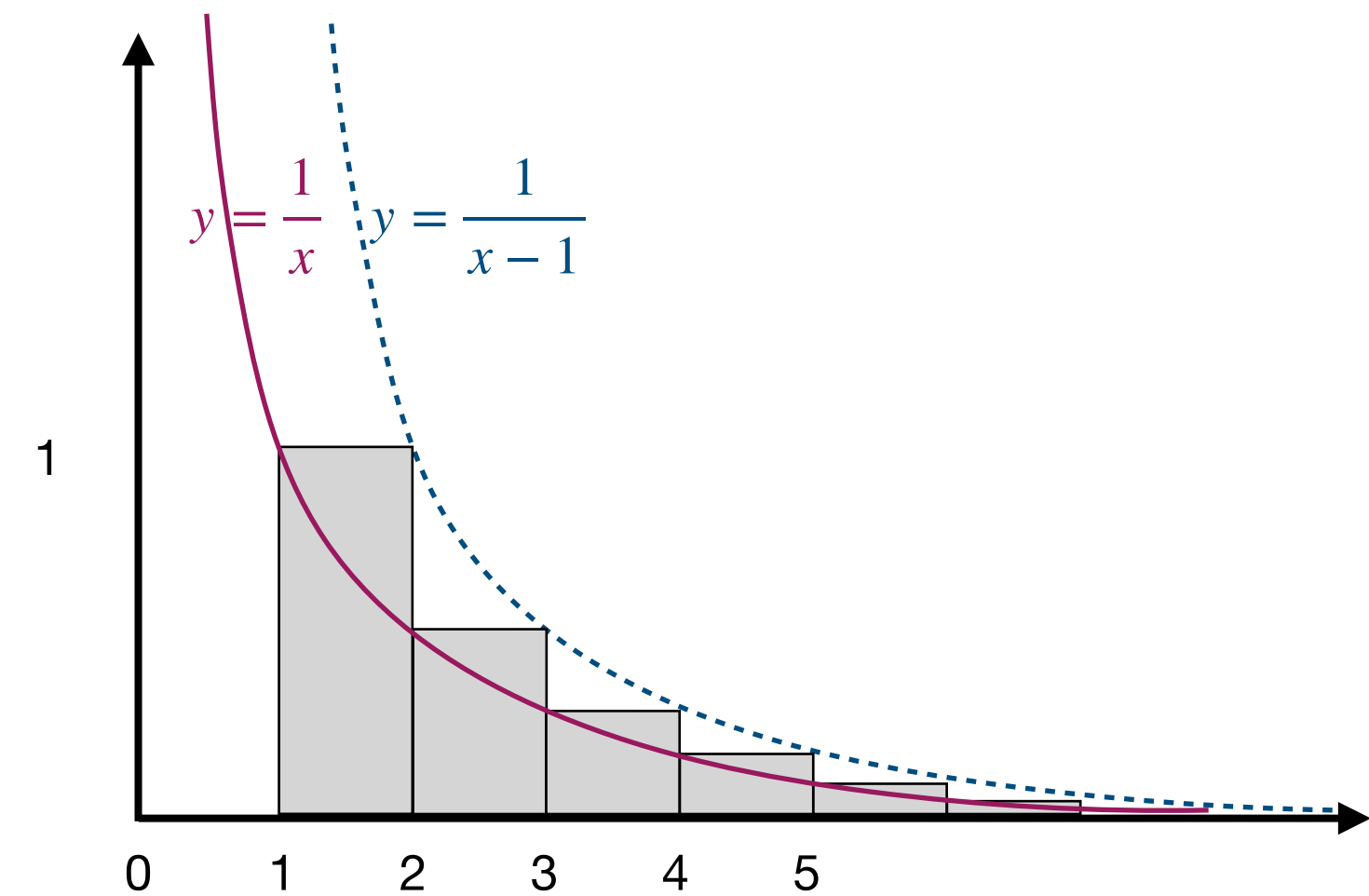
Randomized QuickSort

- Harmonic series

$$\triangleright H_n = \sum_{k=1}^n \frac{1}{k}$$

$$\triangleright \int_1^n \frac{1}{x} dx < \sum_{k=1}^n \frac{1}{k} < 1 + \int_2^n \frac{1}{x-1} dx$$

$$\triangleright \ln n < \sum_{k=1}^n \frac{1}{k} < 1 + \ln n$$





Randomized QuickSort

- Harmonic series

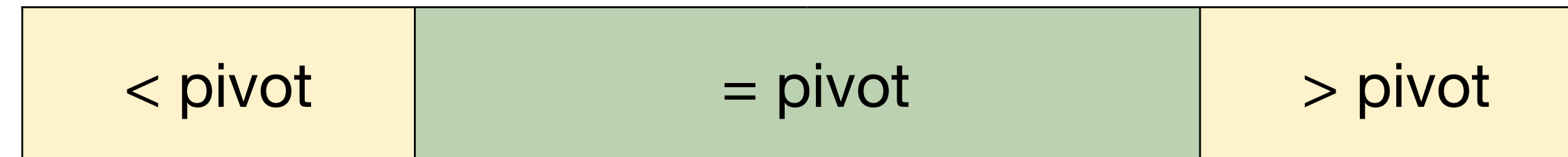
$$\triangleright H_n = \sum_{k=1}^n \frac{1}{k} \sim \ln n$$

- Hence, $\mathbb{E}[X] < \sum_{i=1}^{n-1} \sum_{k=1}^n \frac{2}{k} < 2nH_n < 2n(1 + \ln n) = O(n \lg n)$
- Combined the fact that in the best case (balanced partition each time) randomized quick sort is $\Theta(n \lg n)$, the expected running time is $\Theta(n \lg n)$.
- In fact, runtime of `RndQuickSort` is $O(n \log n)$ with high probability!



A bit more on QuickSort

- What if there are many duplicates?
 - Maintain four regions as we go through the array



- End up with three regions (“<”, “=”, and “>”), and only recurse into two of them (“<” and “>”): the more the duplicates, the less to recurse, and the better the algorithm!



A bit more on QuickSort

- Stop recursion once the array is too small.
 - Recursion has overhead, `QuickSort` is slow on small arrays.
 - Usually using `InsertionSort` for ≈ 10 elements, resulting in fewer swaps, comparisons or other operations on such small arrays.
 - The ideal 'threshold' will vary based on the details of the specific implementation.



A bit more on QuickSort

- “Random pivot selection” and “Median of three” can be combined!
 - ▶ The expected number of comparisons needed to sort n elements with random pivot selection is $2n \ln n = \frac{2n}{\log_2 e} \cdot \log_2 n \approx 1.386n \log_2 n$.
 - ▶ Combining “Median-of-three pivoting” (i.e., randomly selecting three elements and let the median of them to be the pivot) brings this down to about $1.188n \log_2 n$, but at the expense of a three-percent increase in the expected number of **swaps**.
 - ▶ According to Bentley, Jon L.; McIlroy, M. Douglas (1993). "Engineering a sort function". *Software: Practice and Experience*. 23 (11): 1249–1265.



A bit more on QuickSort

- Multiple pivots?
 - ▶ Early studies do not give promising results, until Dual-Pivot variant proposed by Yaroslavskiy in 2009 seems slightly faster.

$< \text{pivot}_1$	$\text{pivot}_1 \leq . \leq \text{pivot}_2$	$> \text{pivot}_2$
--------------------	---	--------------------

- ▶ This variant is used in Java for sorting. (Since Java 7.)
- ▶ According to “Average Case Analysis of Java 7's Dual Pivot Quicksort”. (Best Paper of ESA 2012)



Summary on QuickSort

- A **widely-used** efficient sorting algorithm
- Easy to understand! (divide-and-conquer...)
- Moderately hard to implement correctly. (partition...)
- Harder to analyze. (randomization...)
- Challenging to optimize. (theory and practice...)



The $n \lg n$ sorting algorithms

- QuickSort, MergeSort and HeapSort are all with $O(n \lg n)$, which is better?
 - HeapSort is non-recursive, minimal auxiliary storage requirement (good for embedded system), but with poor **locality of reference**, the access of elements is not linear, resulting many caches being missed! It is the slowest among three algorithms
 - In most (not all) tests, QuickSort turns out to be **faster** than MergeSort. This is because although QuickSort performs 39% more comparisons than MergeSort, but much less movement (copies) of array elements.
 - MergeSort is a stable sorting, and can take advantage of partially pre-sorted input. Further, MergeSort is more efficient at handling slow-to-access sequential media.



External sorting



*External Sorting

- External sorting is required when the data being sorted do not fit into the main memory of a computing device and instead they must reside in the slower external memory, usually a disk drive.
- Since I/O is rather expensive (at the order of 1-10 milliseconds), the overall execution cost may be far dominated by the I/O, the target of algorithm design is to reduce I/Os.
- One challenge to previous internal sorting algorithms is that how to merge big files with small memory!



External merge problem

- **Input:** 2 **sorted** lists (with M and N pages)
- **Output:** 1 merged sorted list (with $M+N$ pages)
- Can we efficiently (in terms of I/O) merge the two lists using a memory buffer of size at least 3?
 - Yes, and by using only $2(M+N)$ I/Os !



Key (Simple) Idea

- To find an element that is no larger than all elements in two lists, one only needs to compare minimum elements from each list

If:

$$A_1 \leq A_2 \leq \dots \leq A_n$$

$$B_1 \leq B_2 \leq \dots \leq B_m$$

Then:

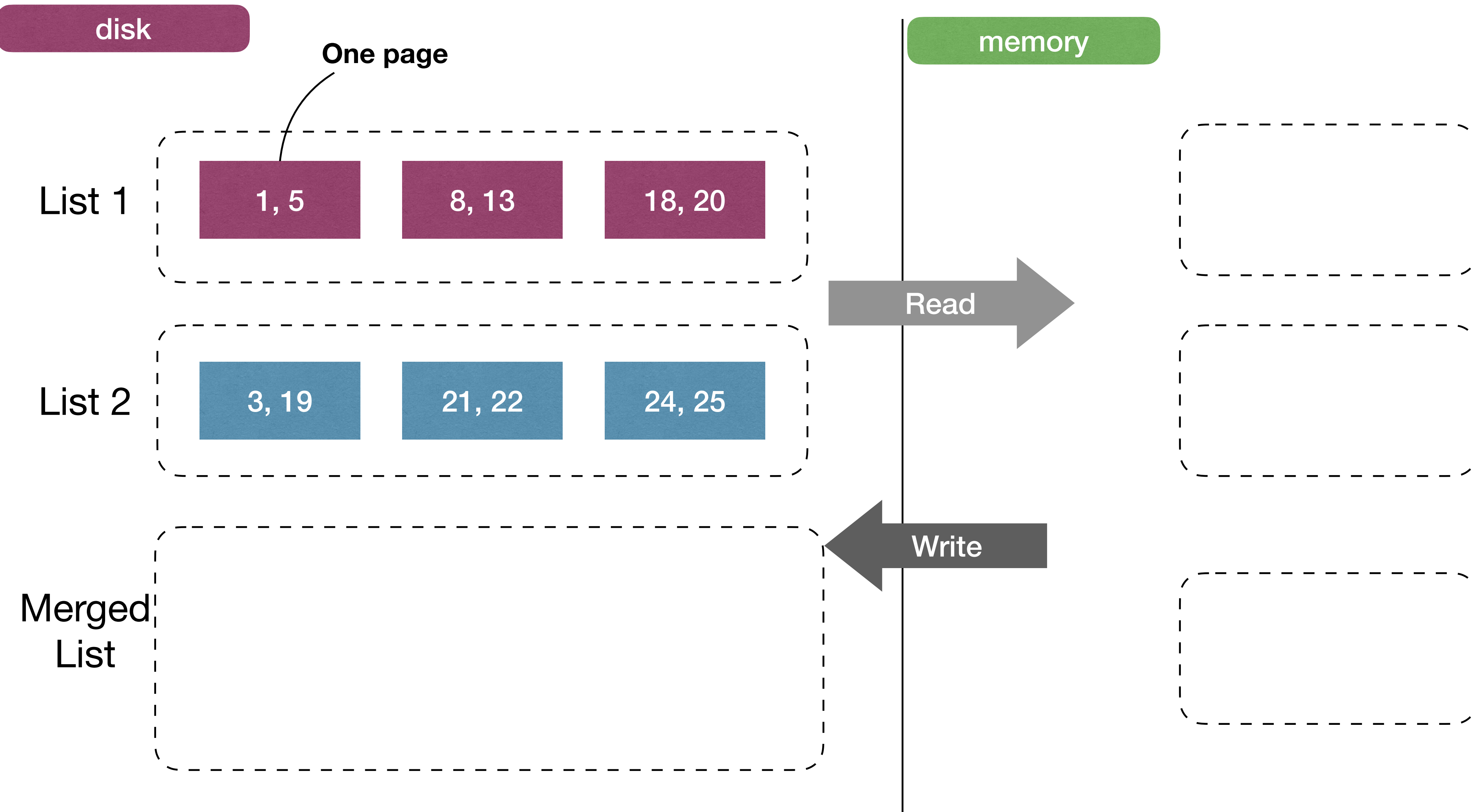
$$\min(A_1, B_1) \leq A_i, \text{ for } 1 \leq i \leq n$$

$$\min(A_1, B_1) \leq B_j, \text{ for } 1 \leq j \leq m$$

- Each time put the current minimum elements back to disk

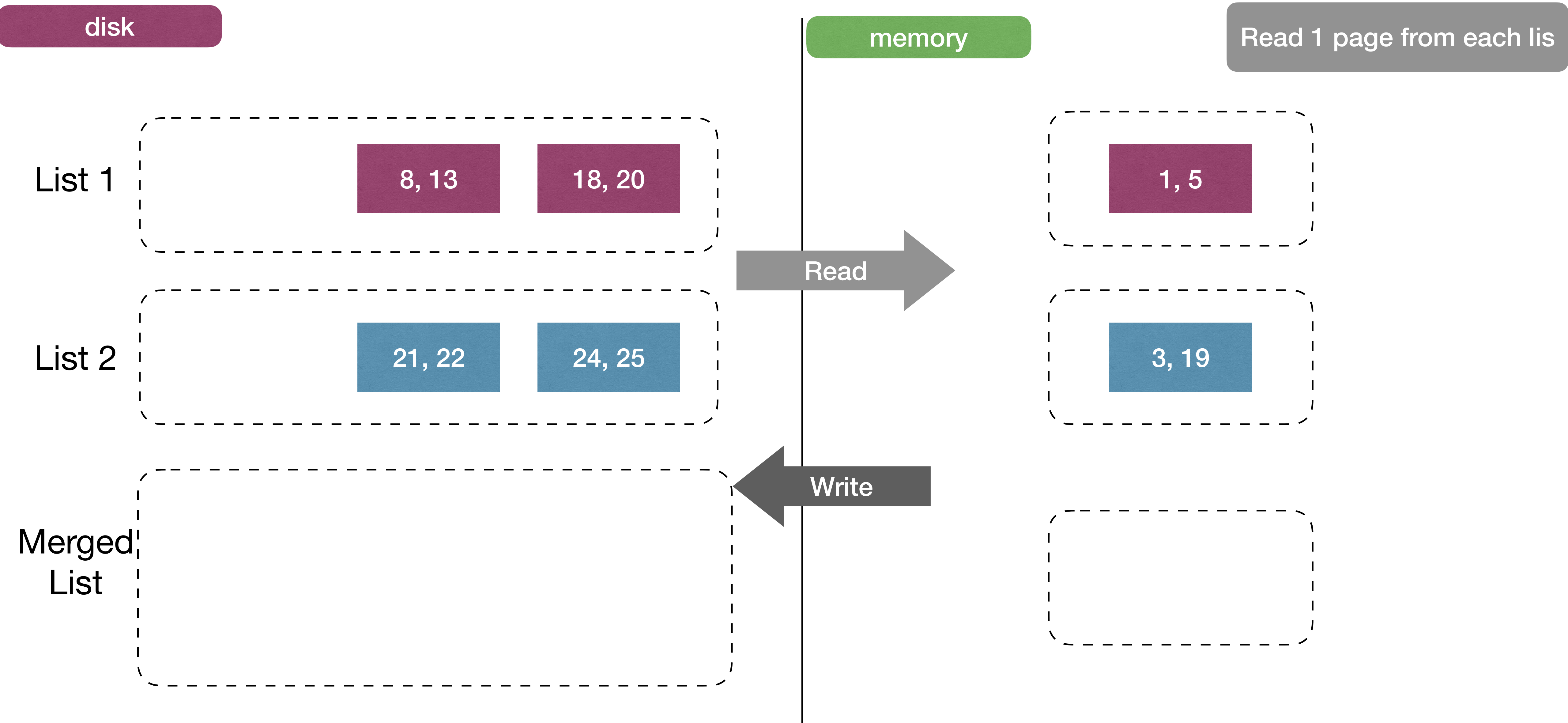


External merge algorithm



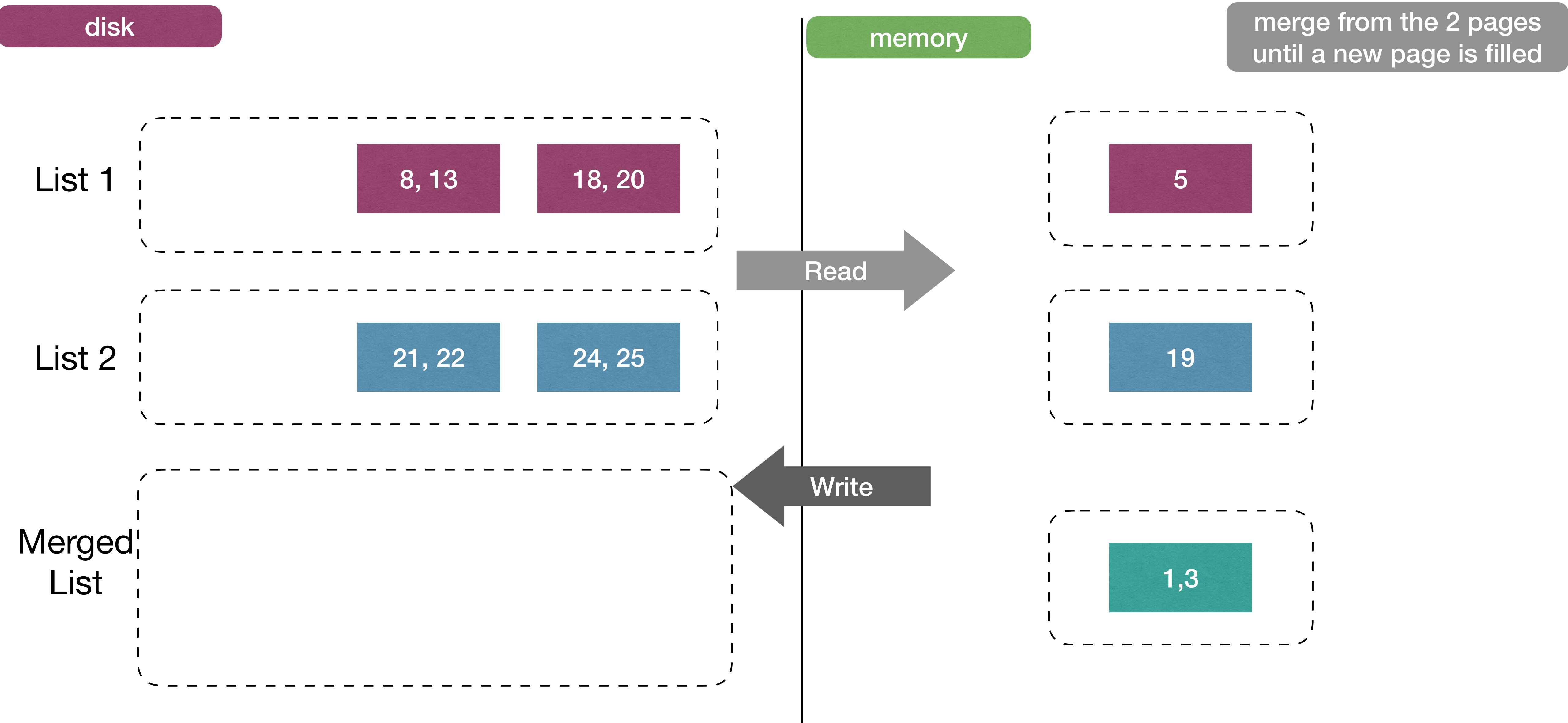


External merge algorithm



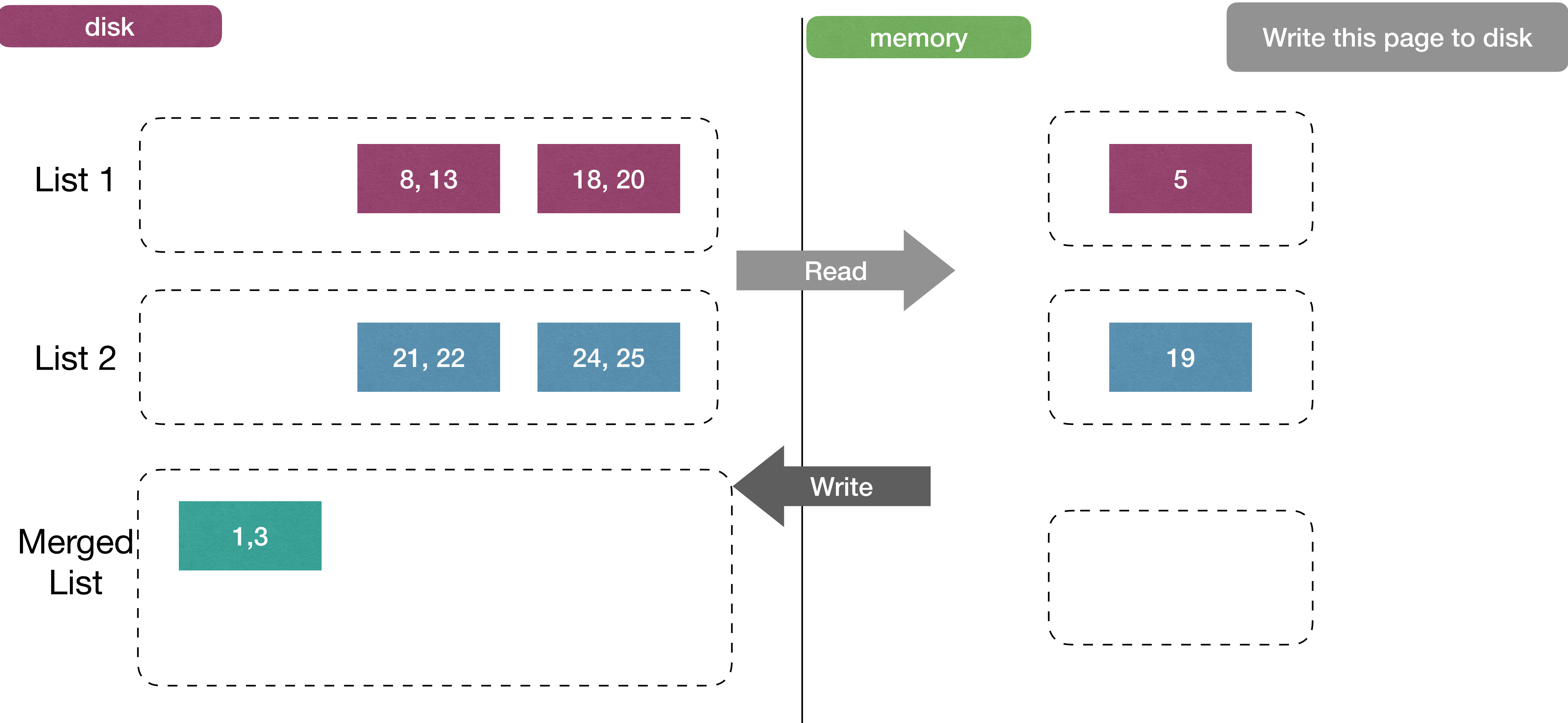


External merge algorithm



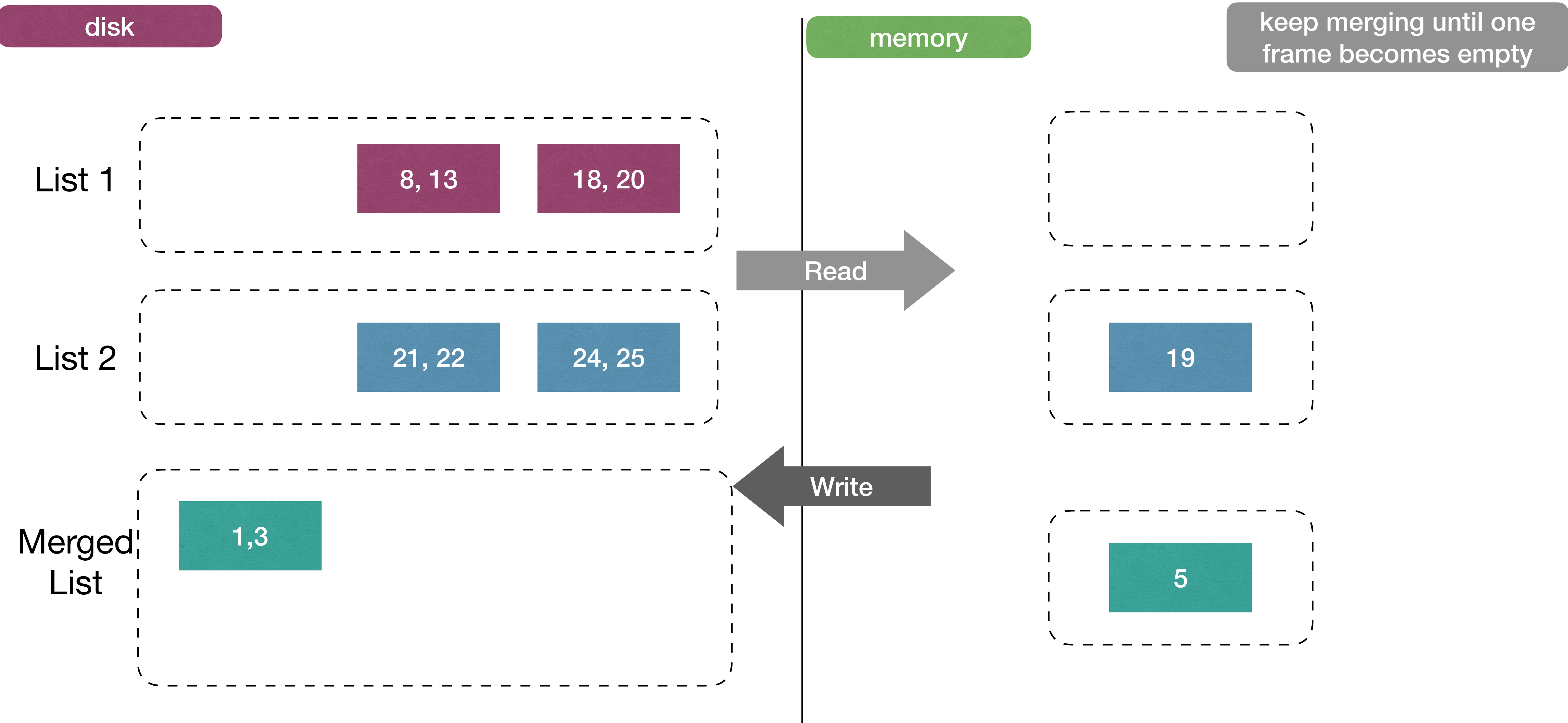


External merge algorithm



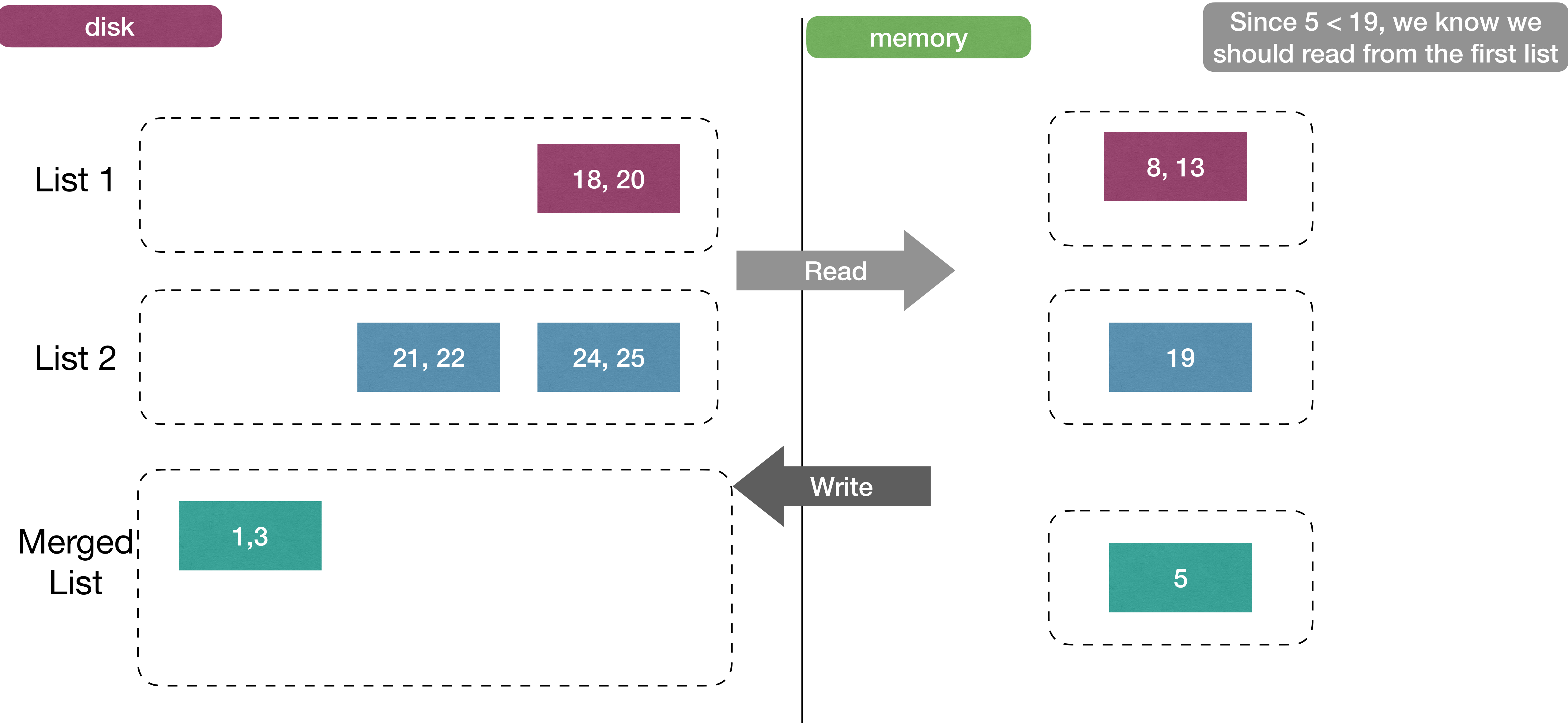


External merge algorithm



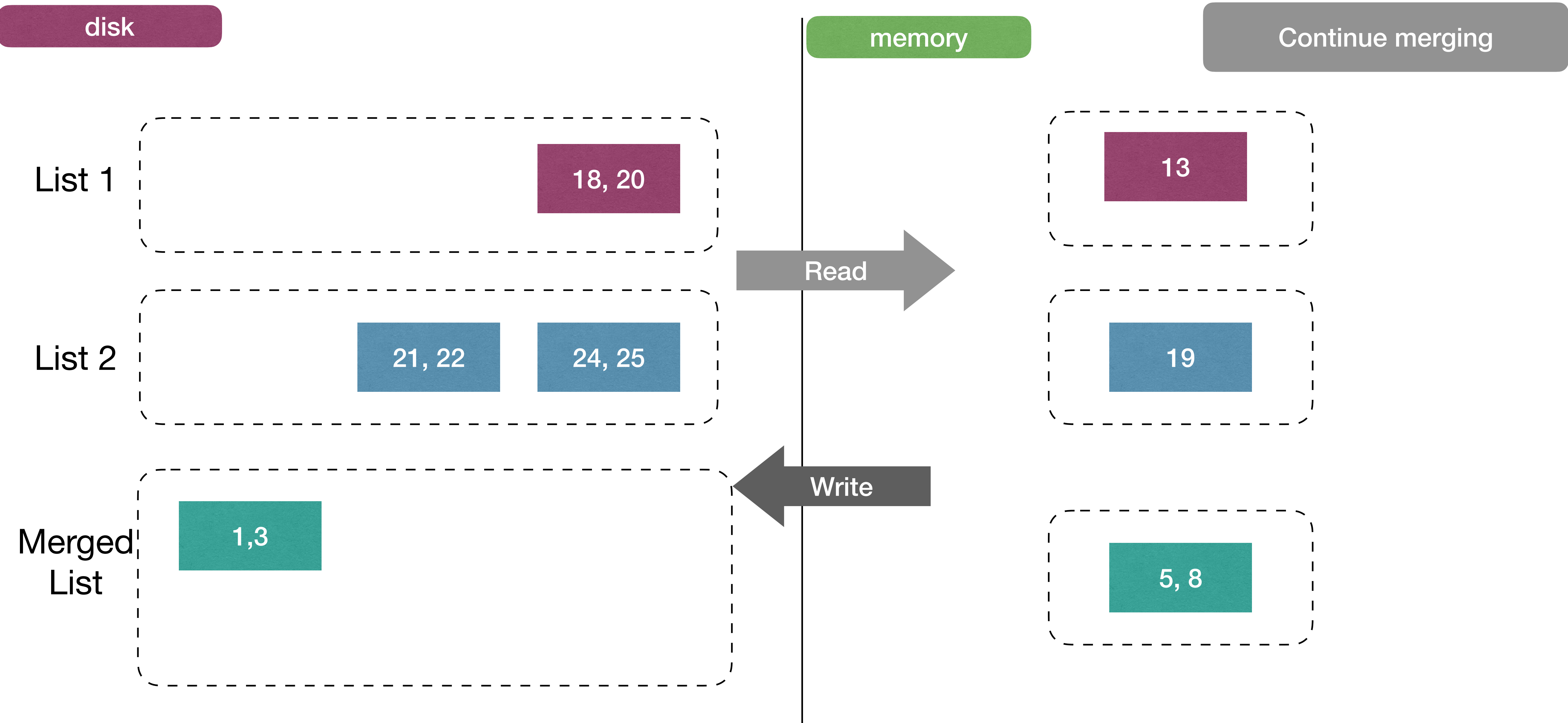


External merge algorithm



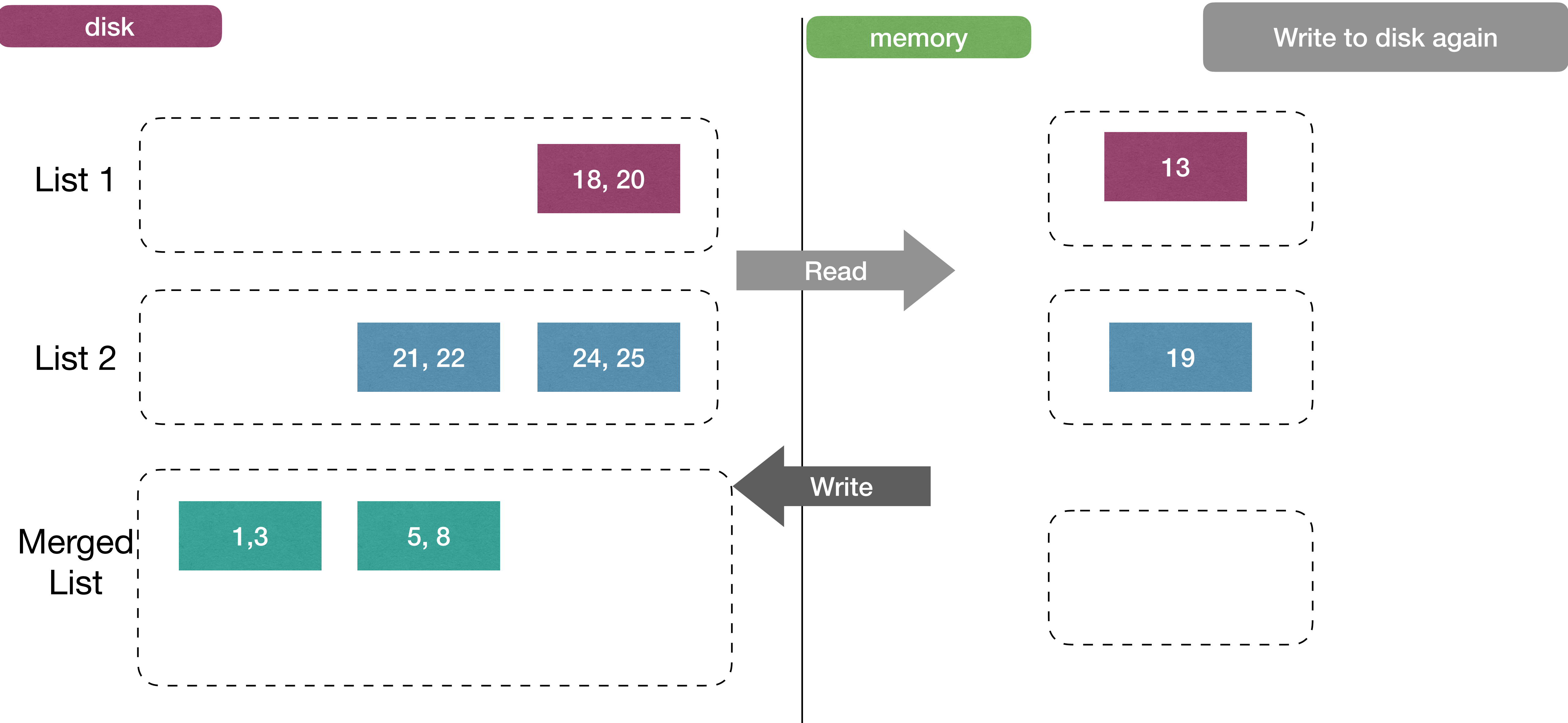


External merge algorithm



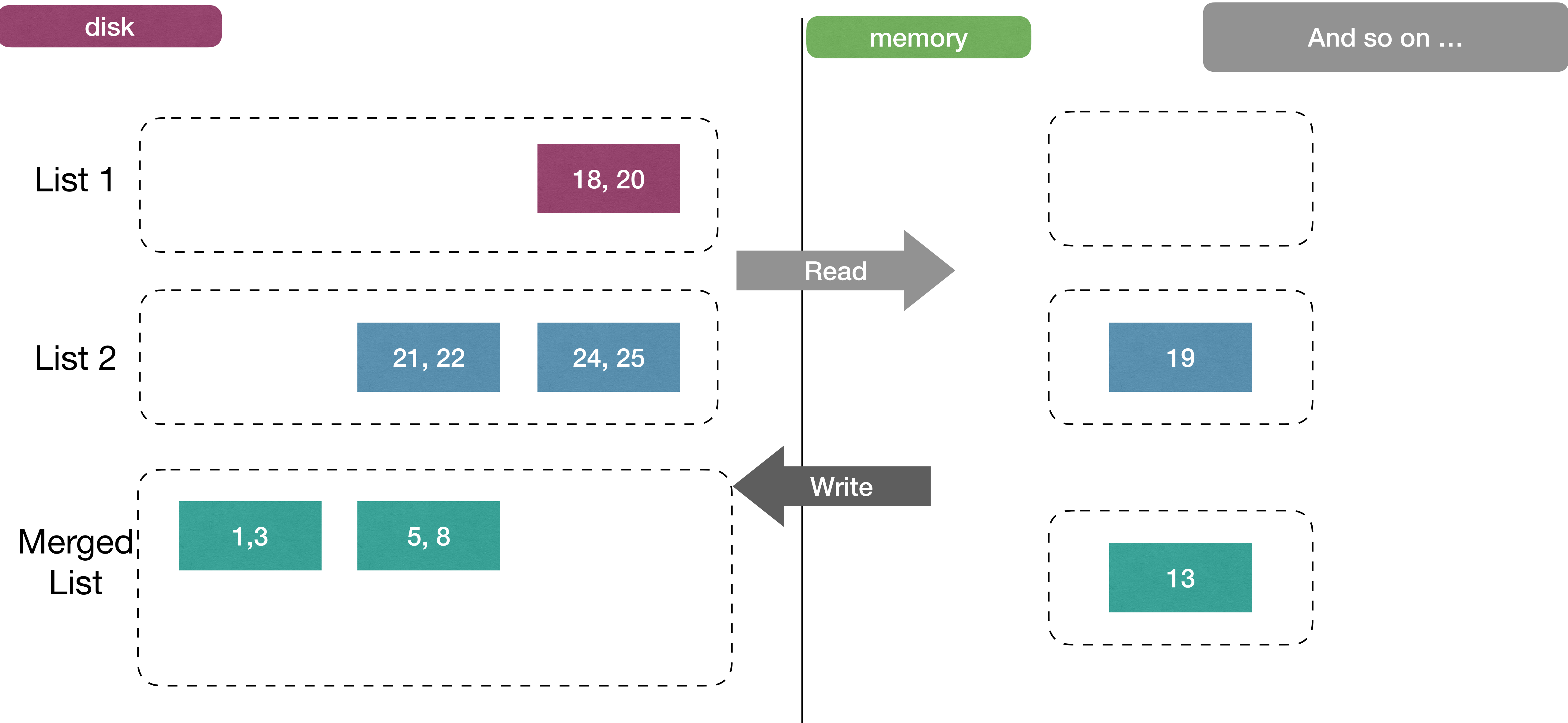


External merge algorithm



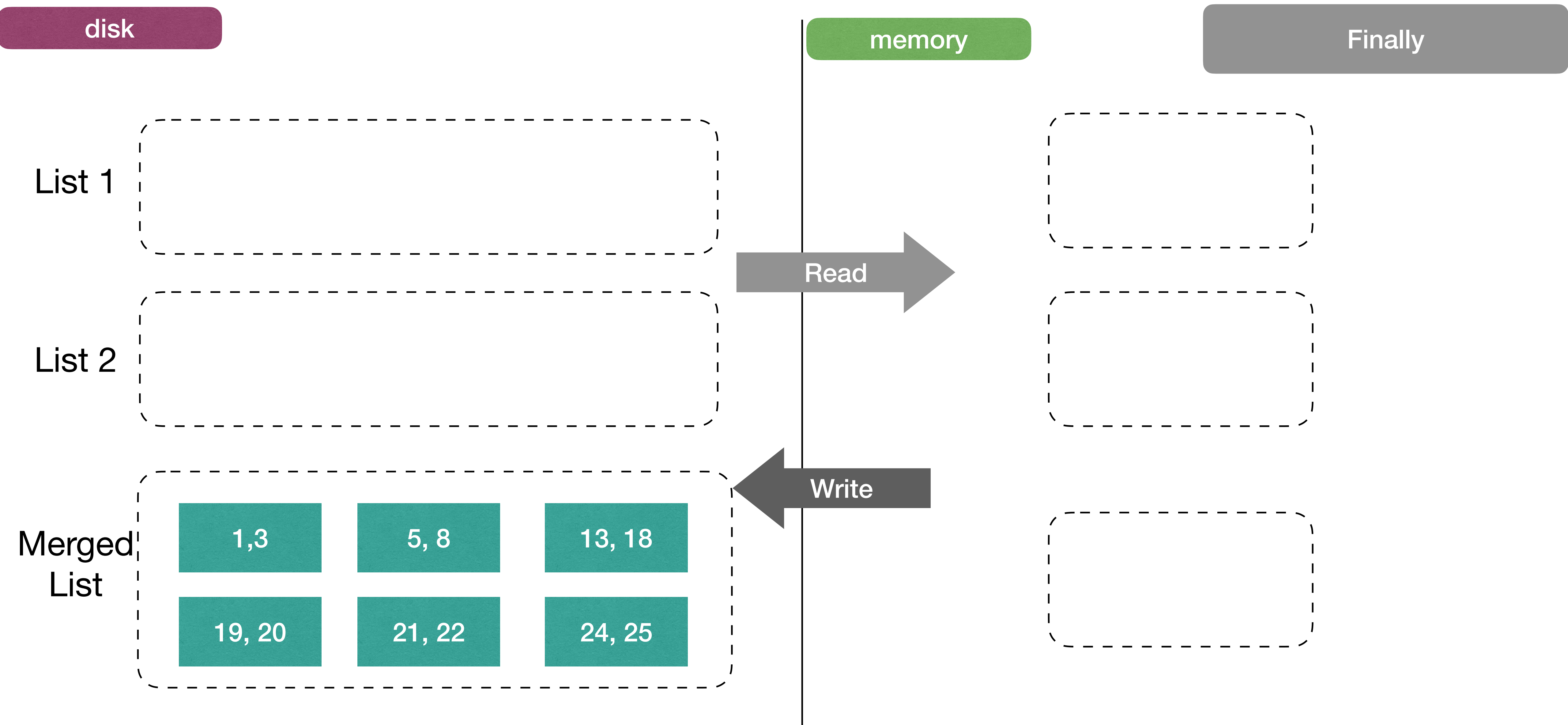


External merge algorithm





External merge algorithm





External merge cost

- We can merge 2 lists of arbitrary length with only 3 buffer pages.
 - I/O cost = $2(M + N)$
- When we have $B+1$ buffer pages, we can merge B lists with the same I/O cost

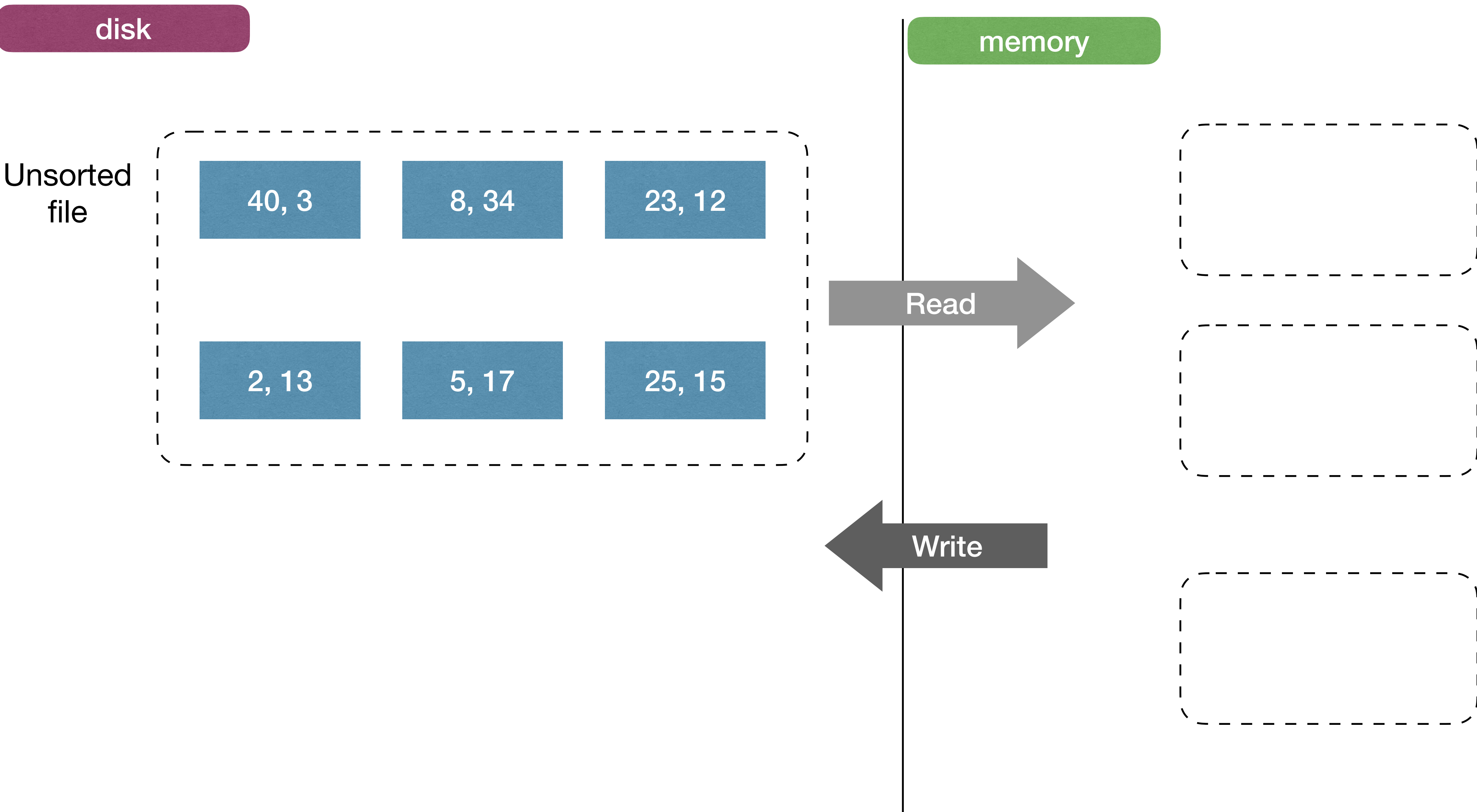


External merge sort

- How to deal with **unsorted** large files?
 - 1. Split into chunks small enough to sort in memory (“runs”)
 - 2. Merge pairs (or groups) of runs using the external merge algorithm
 - 3. Keep merging the resulting runs (each time = a “pass”) until left with one sorted file!



2-Way Sort



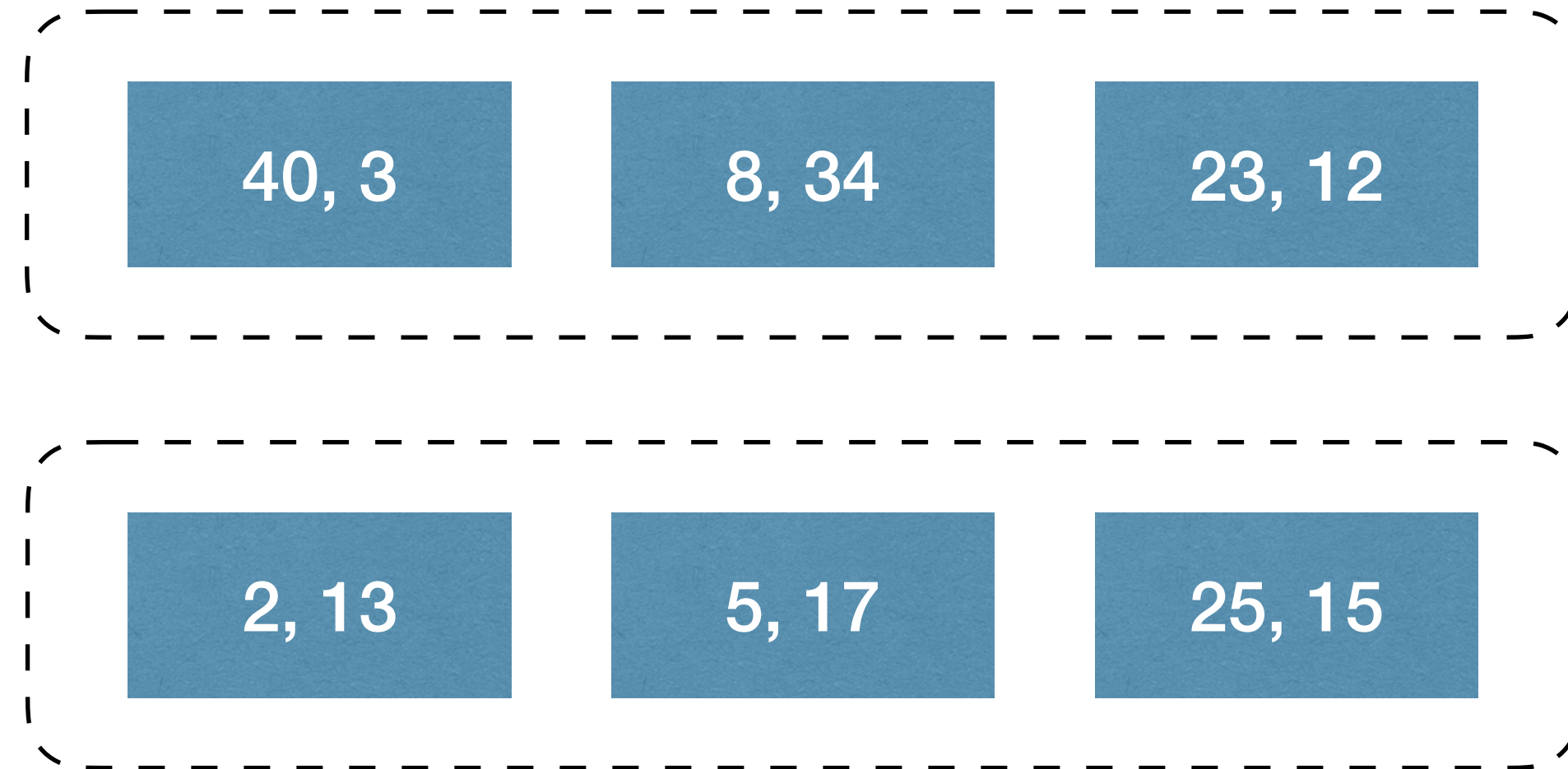


2-Way Sort

disk

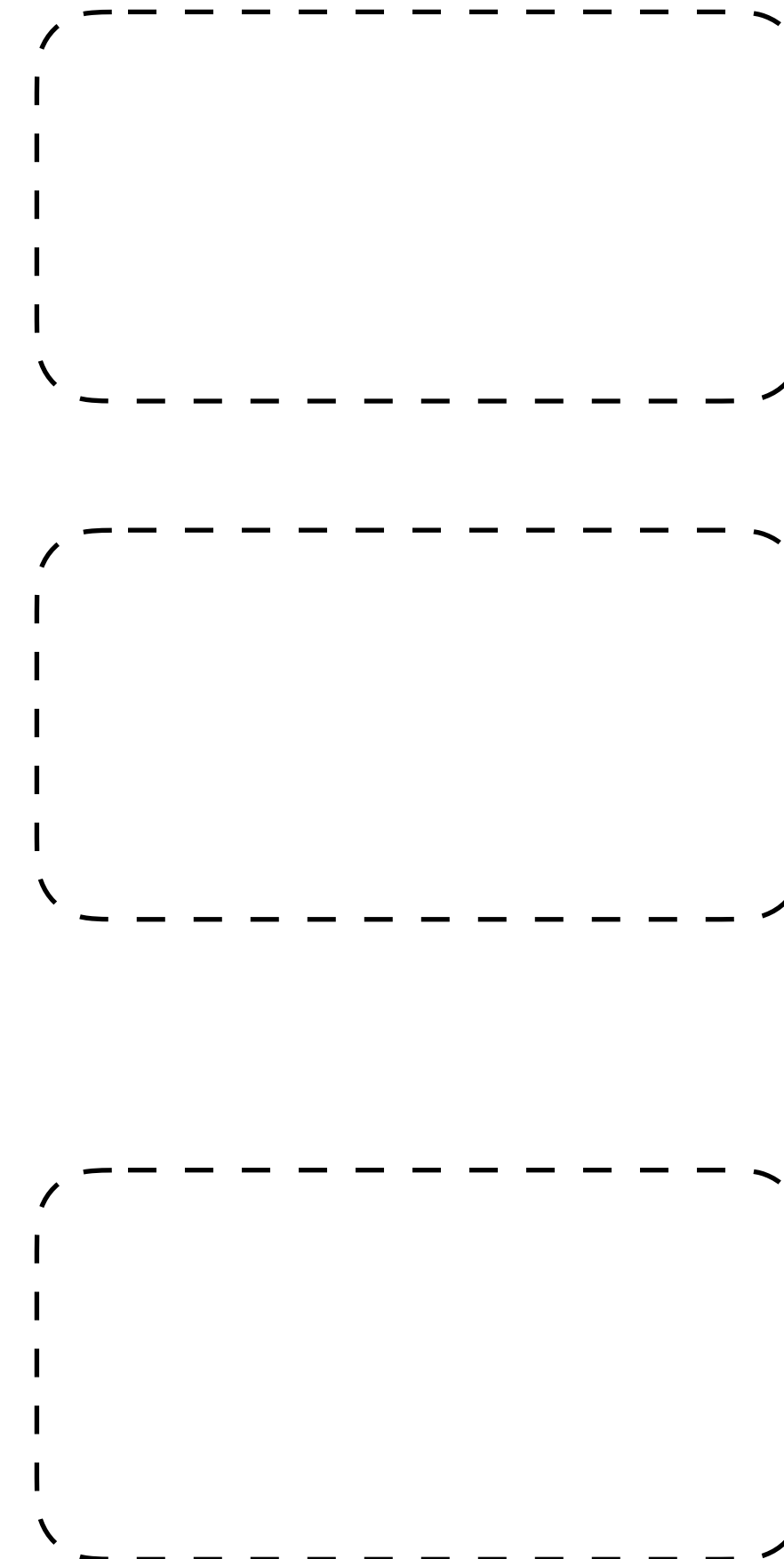
memory

Split into chunks that
fit in memory



Read

Write



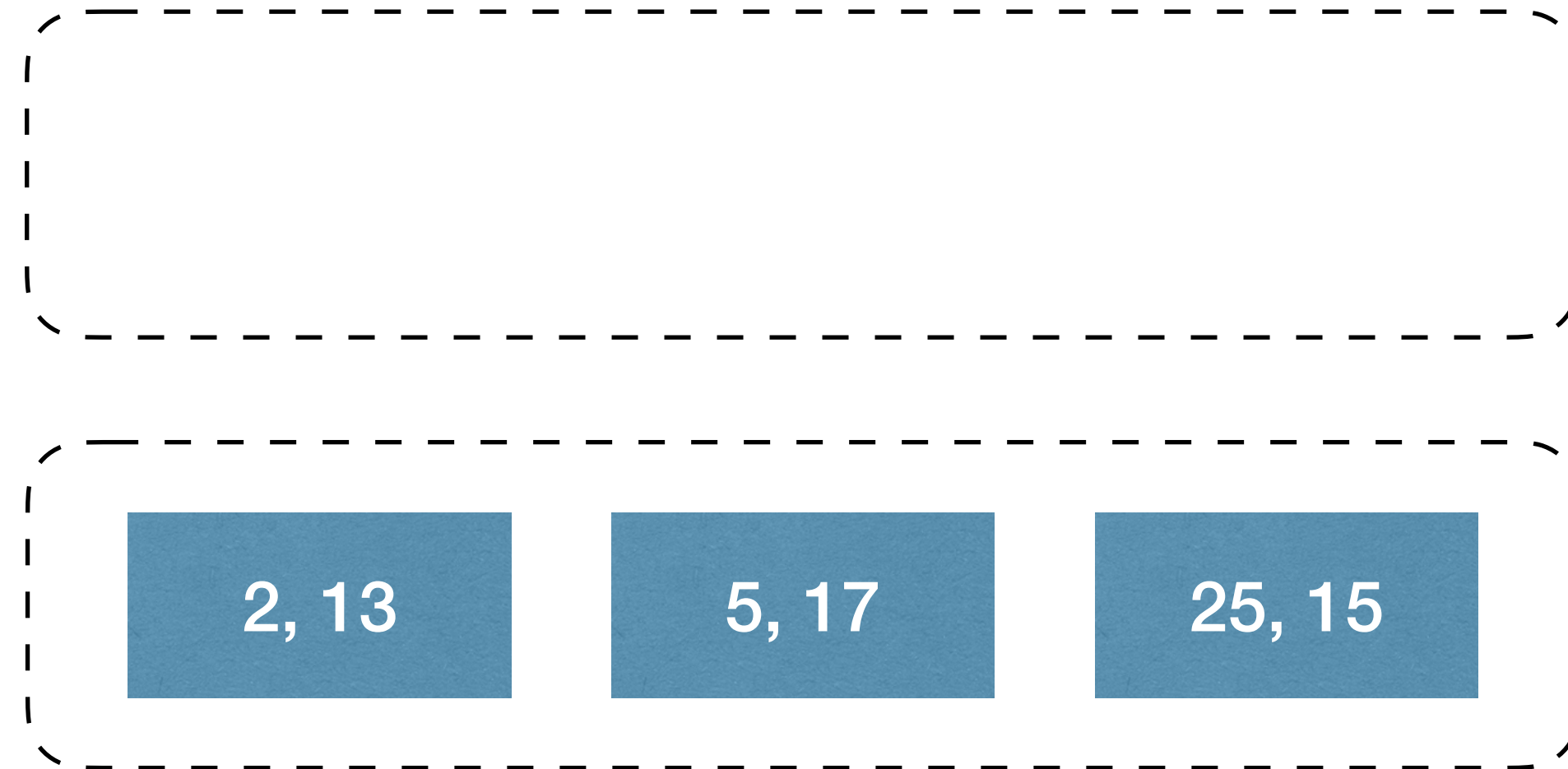


2-Way Sort

disk

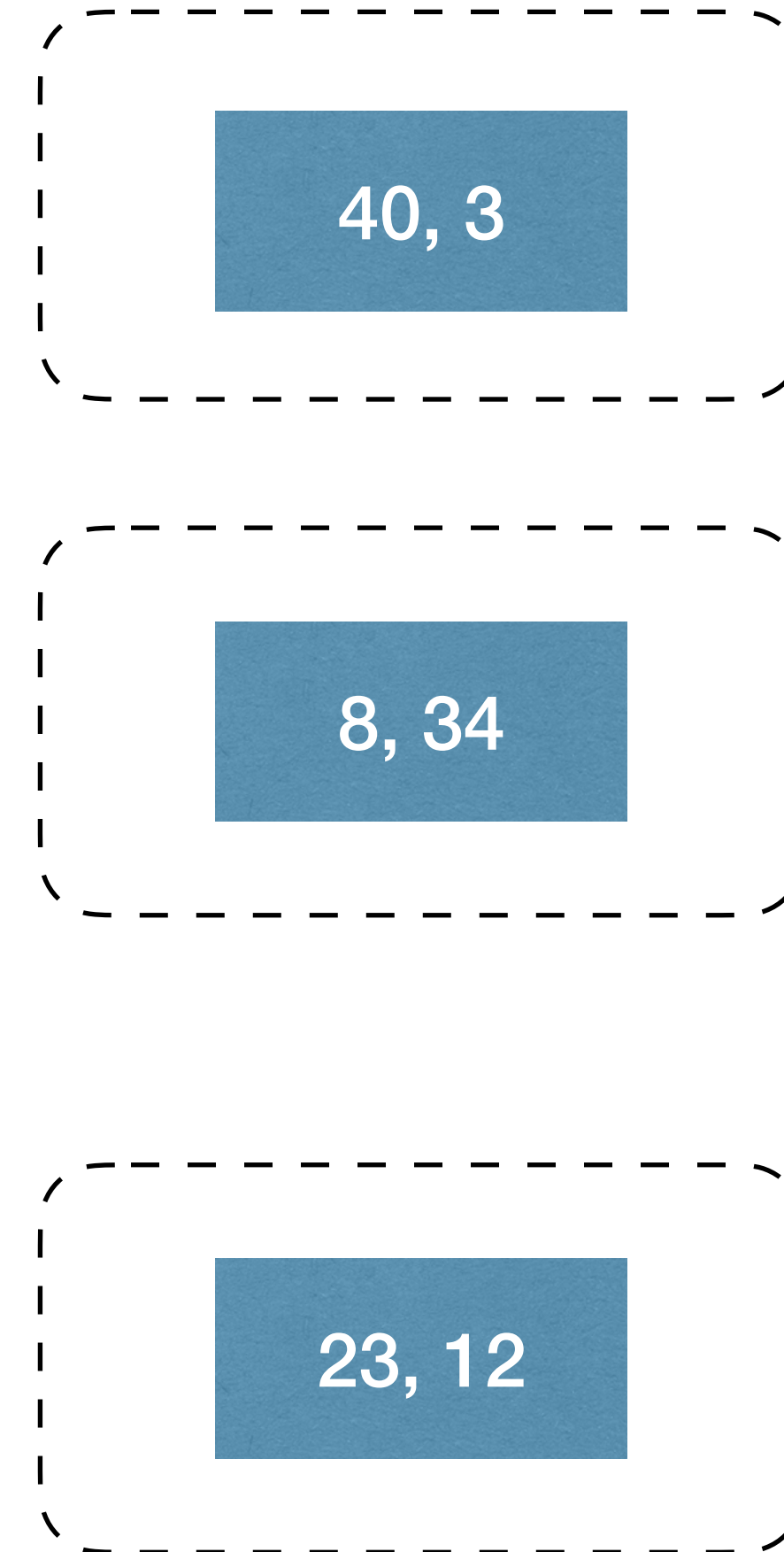
memory

read each chunk in memory



Read

Write



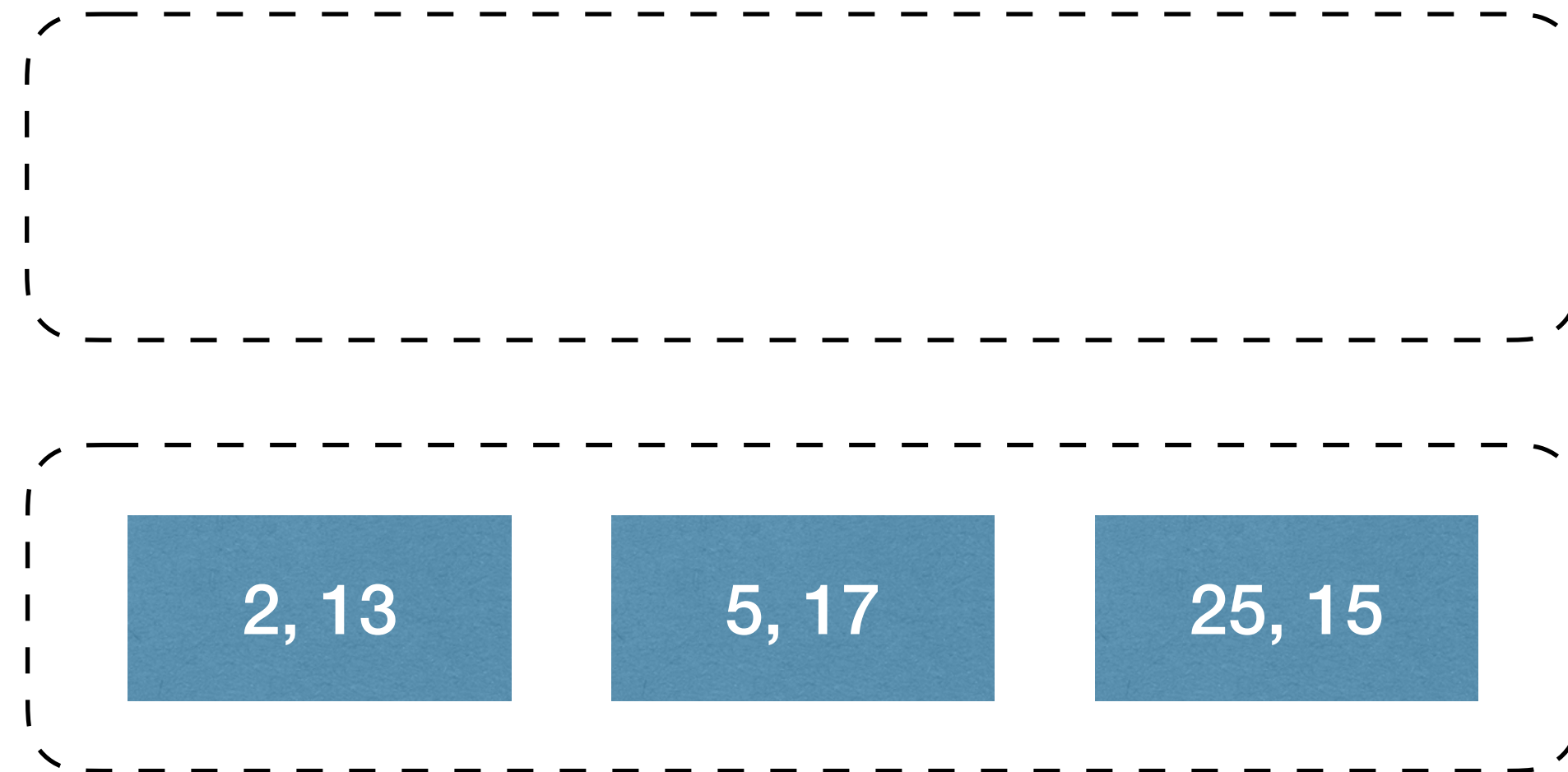


2-Way Sort

disk

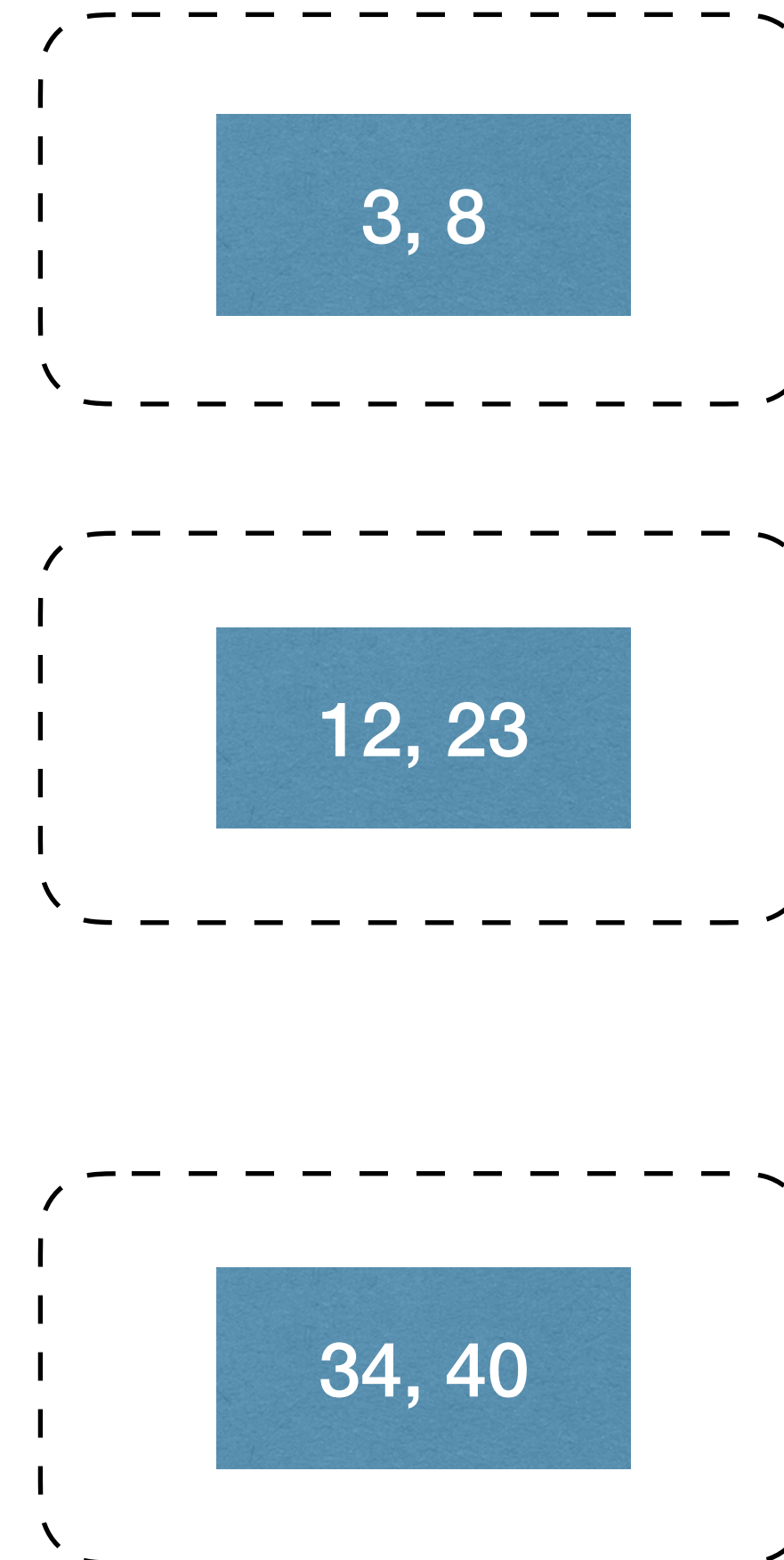
memory

sort in memory



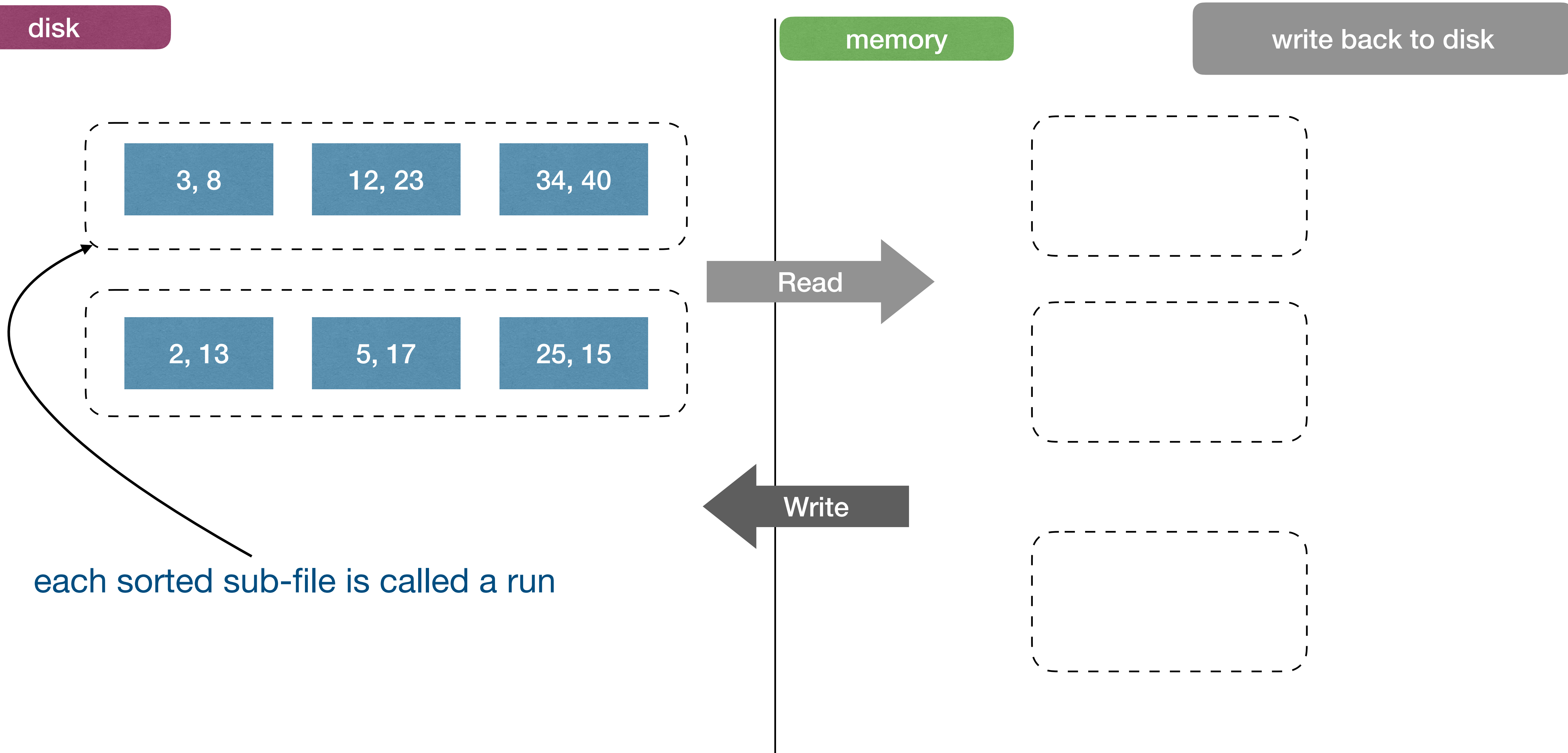
Read

Write





2-Way Sort



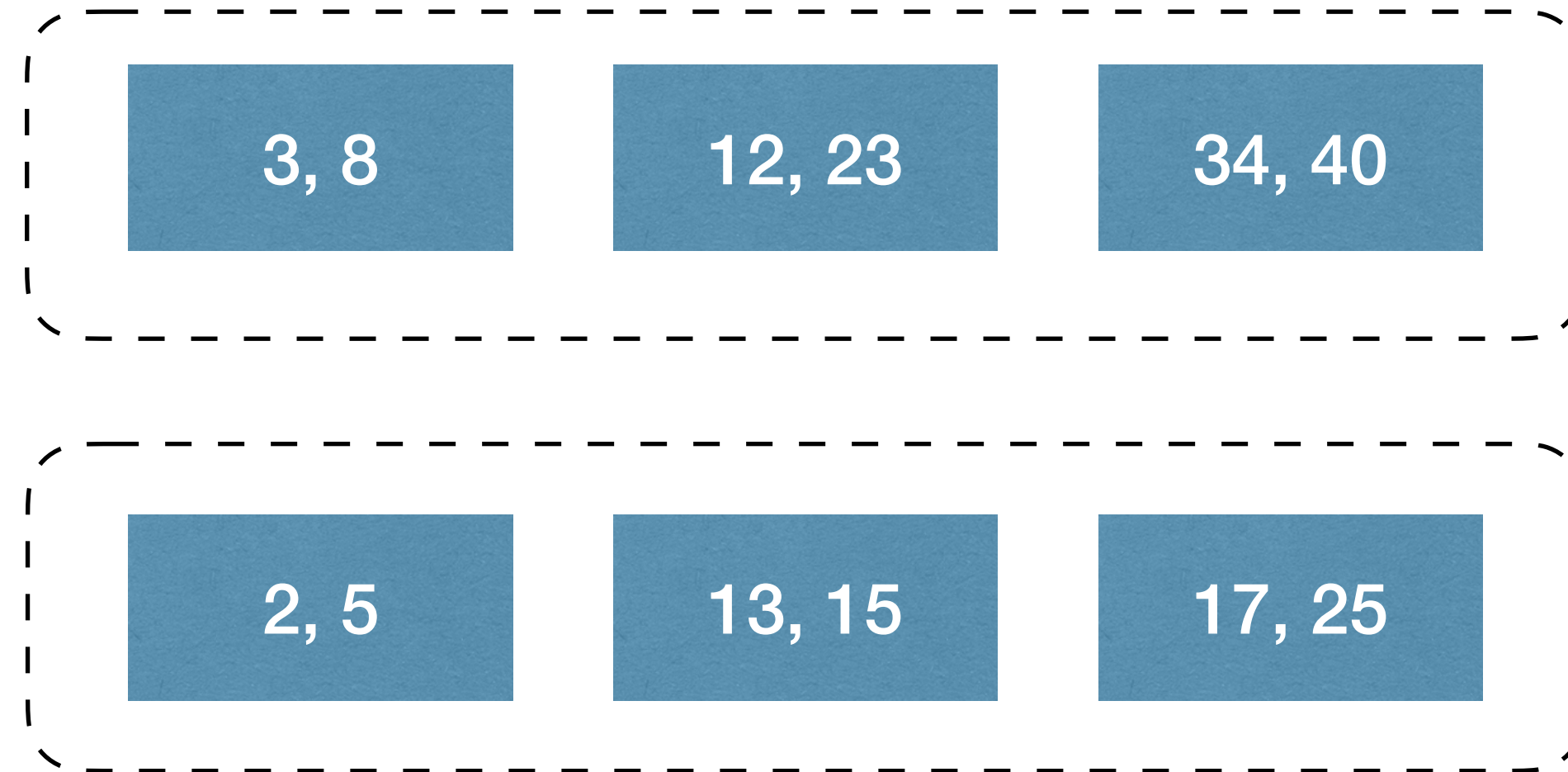


2-Way Sort

disk

memory

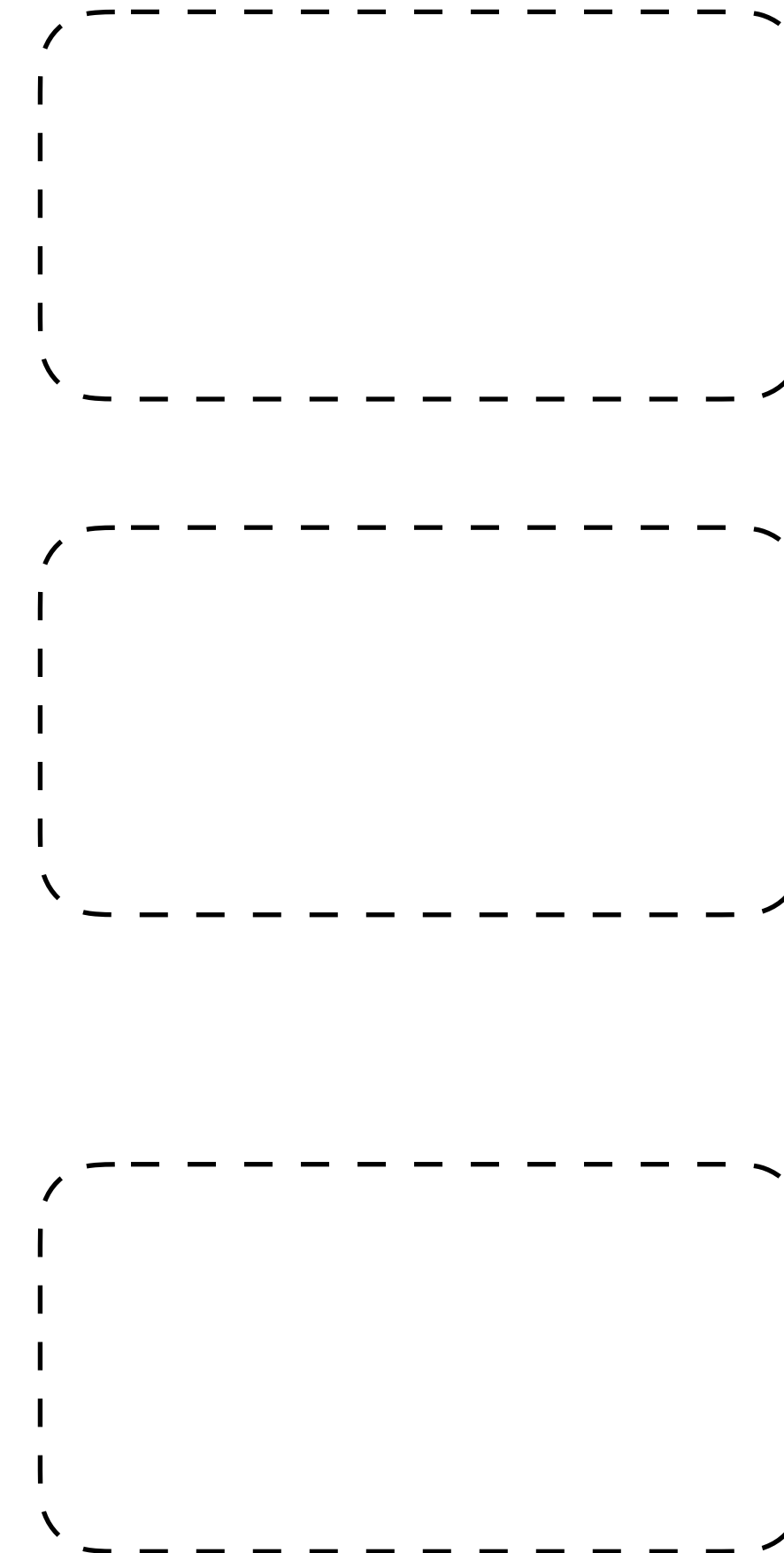
same for the other chunk



Read

Write

Now we have 2 runs





2-Way Sort

disk

memory

3, 8

12, 23

34, 40

2, 5

13, 15

17, 25

Read

Write

final step: use the external
sort merge algorithm to
merge the 2 runs



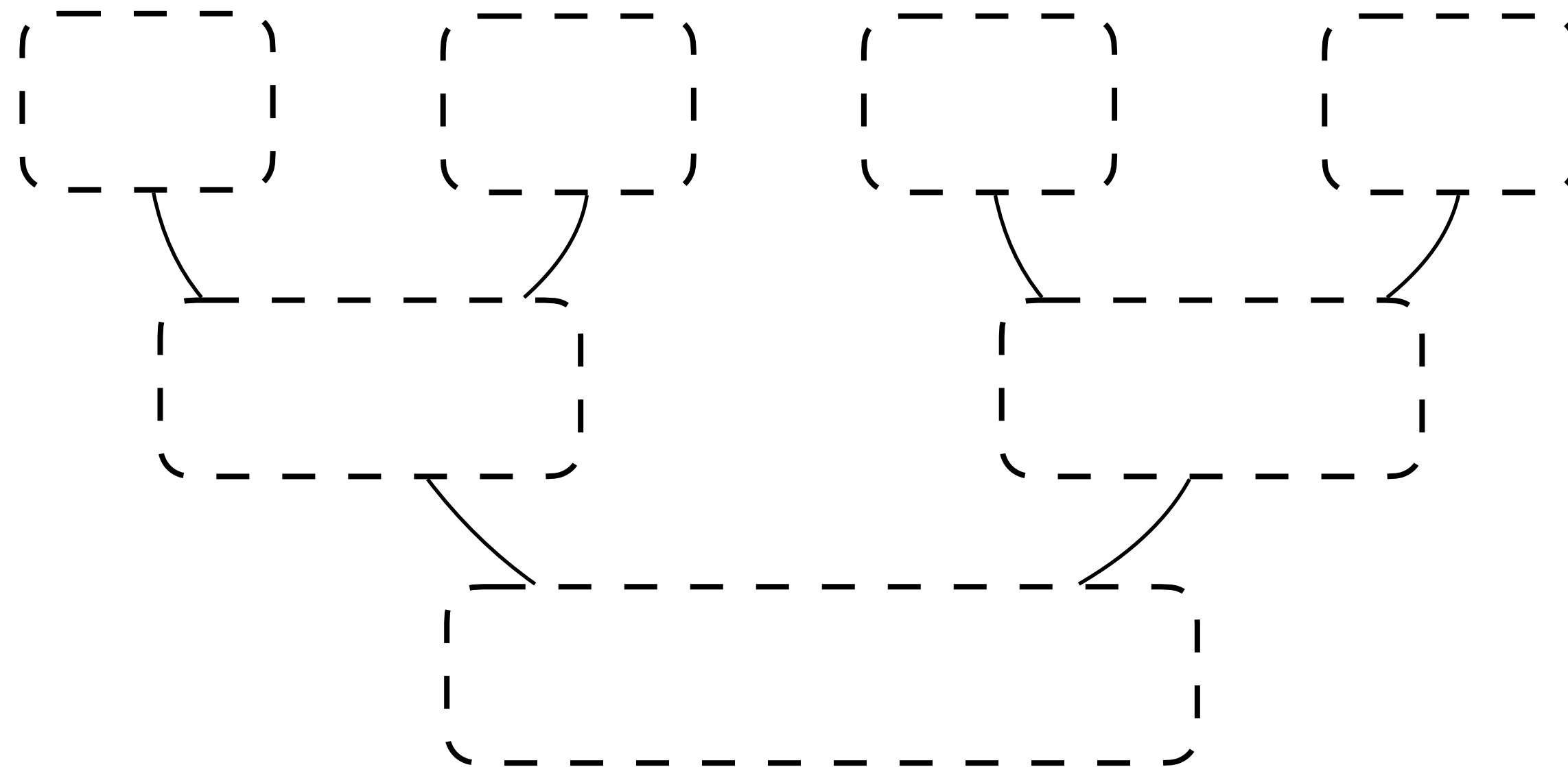
Calculating the I/O cost

- In our example there are 3 buffer pages, and 6 pages
- Pass 0: creating the runs
 - 1 read + 1 write for every page
 - total cost = $6 * (1 + 1) = 12$ I/Os
- Pass 1: external merge sort
 - total cost = $2 * (3 + 3) = 12$ I/Os
- So 24 I/Os in total



I/O Cost: Simplified Version

- Assume for now that we initially create N runs, each run consisting of a **single** page



pass 0: N runs, each 1 page

pass 1: merge into $N/2$ runs, each two pages

pass 2: merge into $N/4$ runs, each with 4 pages

- We need $\lceil \log_2 N \rceil + 1$ passes to sort the whole file, each pass needs $2N$ I/Os
- Total I/O cost = $2N(\lceil \log_2 N \rceil + 1)$



Can we do better?

- The 2-way merge algorithm only uses 3 buffer pages
- What if we have more available memory?
 - Use as much of the available memory as possible in every pass
 - Reducing the number of passes reduces I/O



External sort: I/O cost

- Suppose we have $B \geq 3$ buffer pages available

1. Increase length of initial runs

- At the beginning, we can split the N pages into runs of length B and sort these in memory
- IO cost:

$$2N(\lceil \log_2 N \rceil + 1)$$

Starting with runs
of length 1



$$2N(\lceil \log_2 \frac{N}{B} \rceil + 1)$$

Starting with runs
of length B



External sort: I/O cost

- Suppose we have $B \geq 3$ buffer pages available

2. Perform a $(B - 1)$ –way merge.

- ▶ On each pass, we can merge groups of $(B - 1)$ runs at a time, instead of merging pairs of runs!
- ▶ IO cost:

$$2N(\lceil \log_2 N \rceil + 1)$$

Starting with runs
of length 1

$$2N(\lceil \log_2 \frac{N}{B} \rceil + 1)$$

Starting with runs
of length B

$$2N(\lceil \log_{B-1} \frac{N}{B} \rceil + 1)$$

Performing $(B - 1)$ –way merge



Further reading

- [CLRS] Ch.7, Appendix C on probability theory
- [Weiss] Ch. 7 (7.4, 7.12)
- [Deng] Ch.12 (12.3)
- [TAOCP] Ch.5 (5.2.1 in vol. 3)

