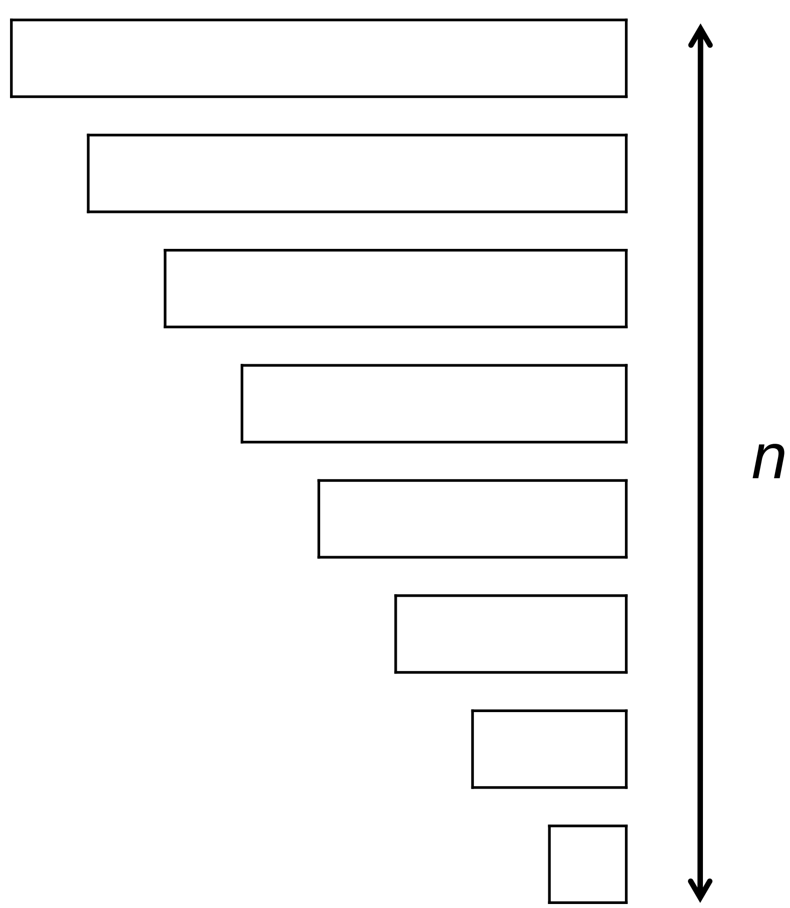
Sorting

Divide and Conquer

Searching an *n*-element Array

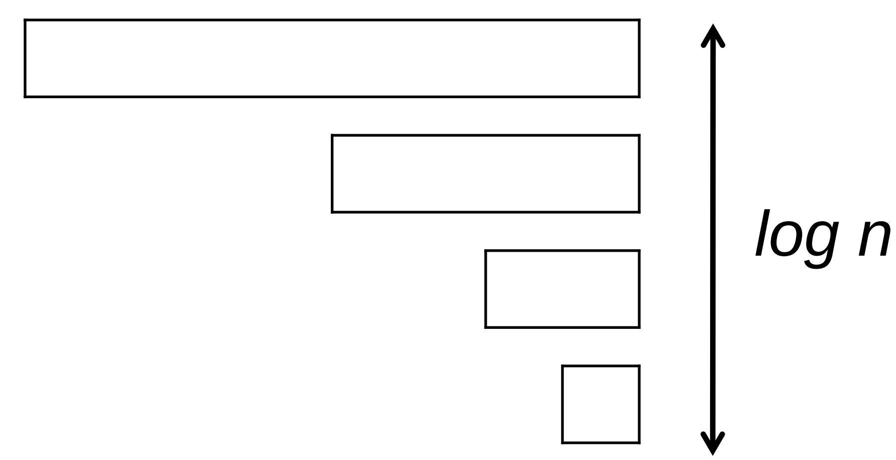
Linear Search

- Check an element
- If not found,
  search an (n-1)–element array



Binary Search

- Check an element
- If not found,
  search an (n/2)–element array

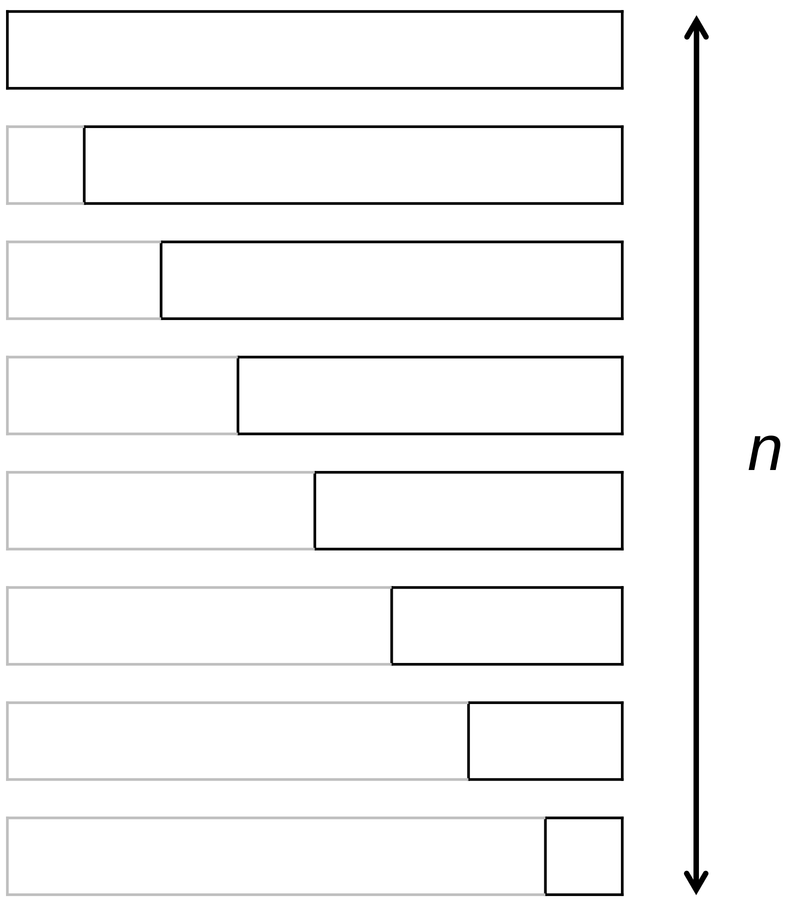


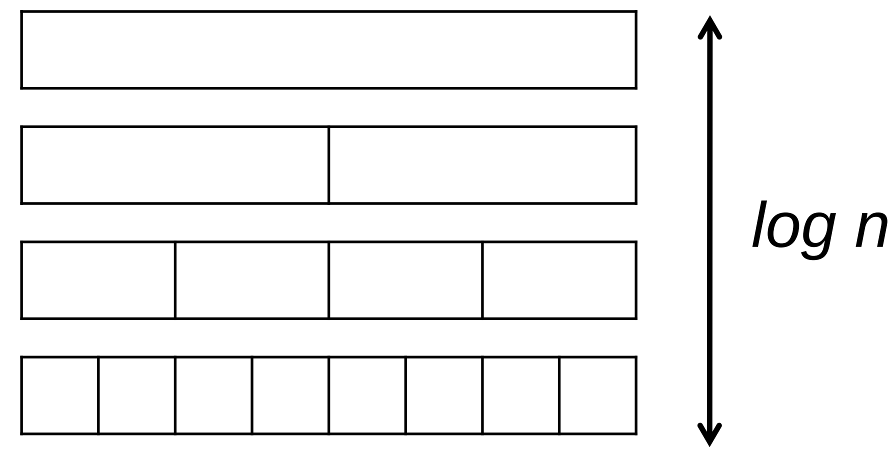
Huge benefit by
dividing problem
(in **half**)

O(n) ⇒ *O(log n)*

Sorting an *n*-element Array

- Can we do the same for sorting an array?
- This time, we need to work on **two half-problems**
  - and combine their results





- This is a general technique called

  divide and conquer

Term variously attributed to Ceasar, Macchiavelli, Napoleon, Sun Tzu, *and many others*

Sorting an *n*-element Array

	Naïve algorithm	Divide and Conquer algorithm
Searching	Linear search *O(n)*	Binary search *O(log n)*
Sorting	Selection Sort $O(n^2)$	??? sort *O(??)*

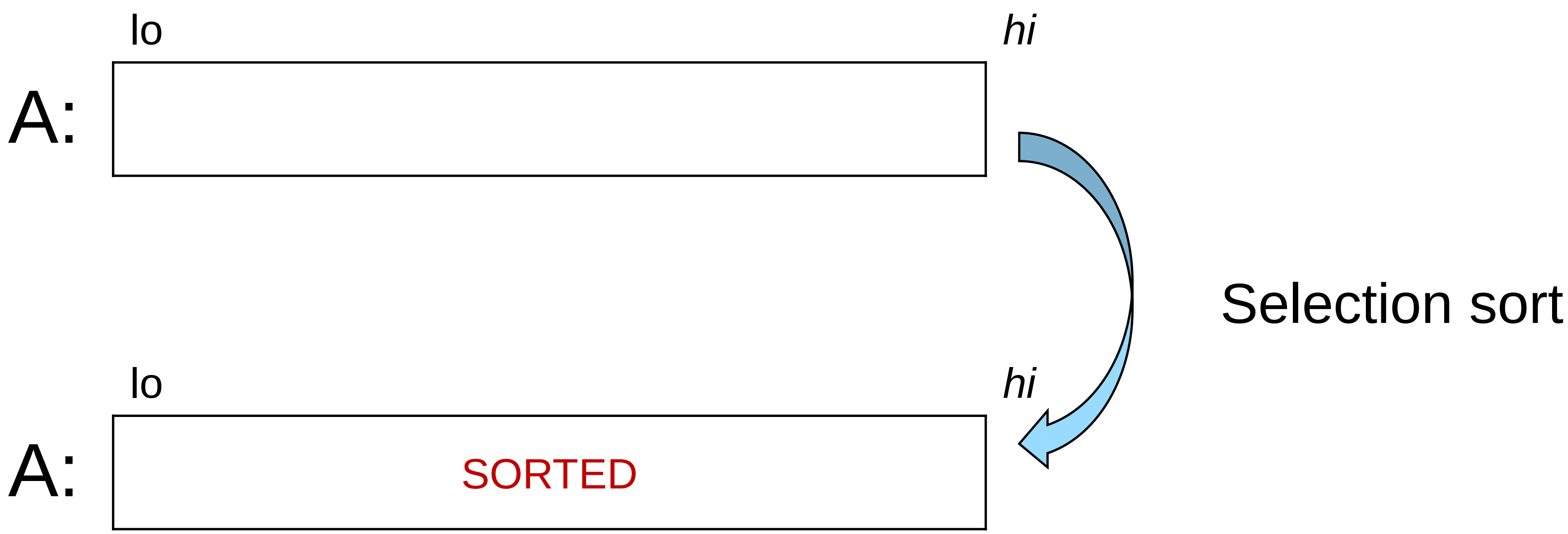
Recall Selection Sort

```
void selection_sort(int[] A, int lo, int hi)
//@requires 0 <= lo && lo <= hi  && hi <= \length(A);
//@ensures is_sorted(A, lo, hi);
{
  for (int i = lo; i < hi; i++)
  //@loop_invariant lo <= i && i <= hi;
  //@loop_invariant is_sorted(A, lo, i);
  //@loop_invariant le_segs(A, lo, i, A, i, hi);
  {
    int min = find_min(A, i, hi);
    swap(A, i, min);
  }
}
```

$O(n^2)$

Towards Mergesort

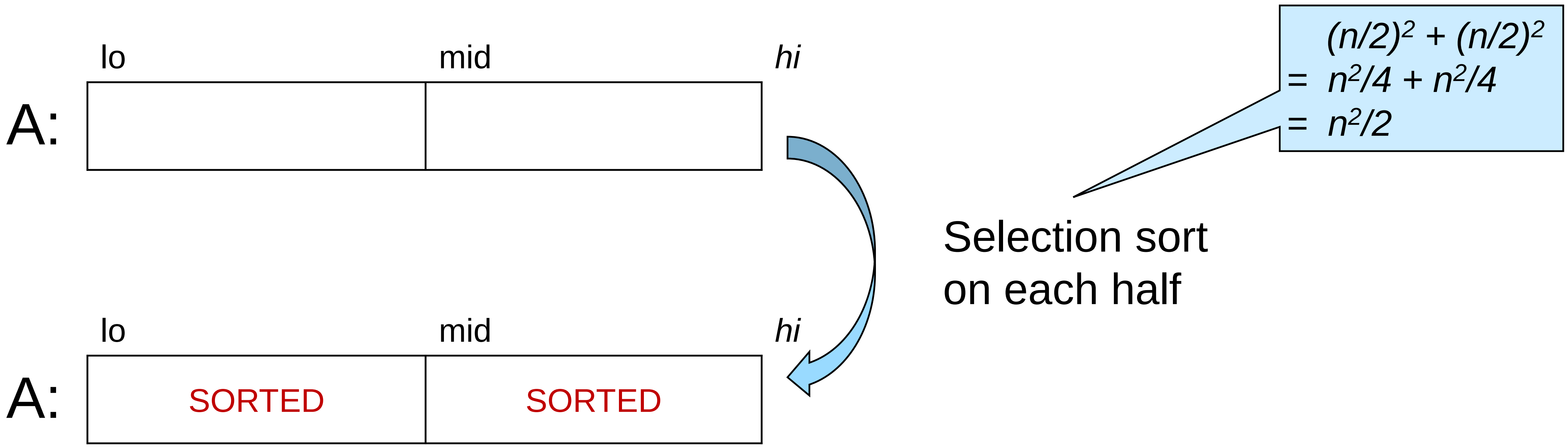
Using Selection Sort



- If $hi - lo = n$
  - ➢ the length of array segment $A[lo, hi)$
  - ○ cost is $O(n^2)$
  - ○ let's say n^2

- But $(n/2)^2 = n^2/4$
  - ○ What if we sort the two halves of the array?

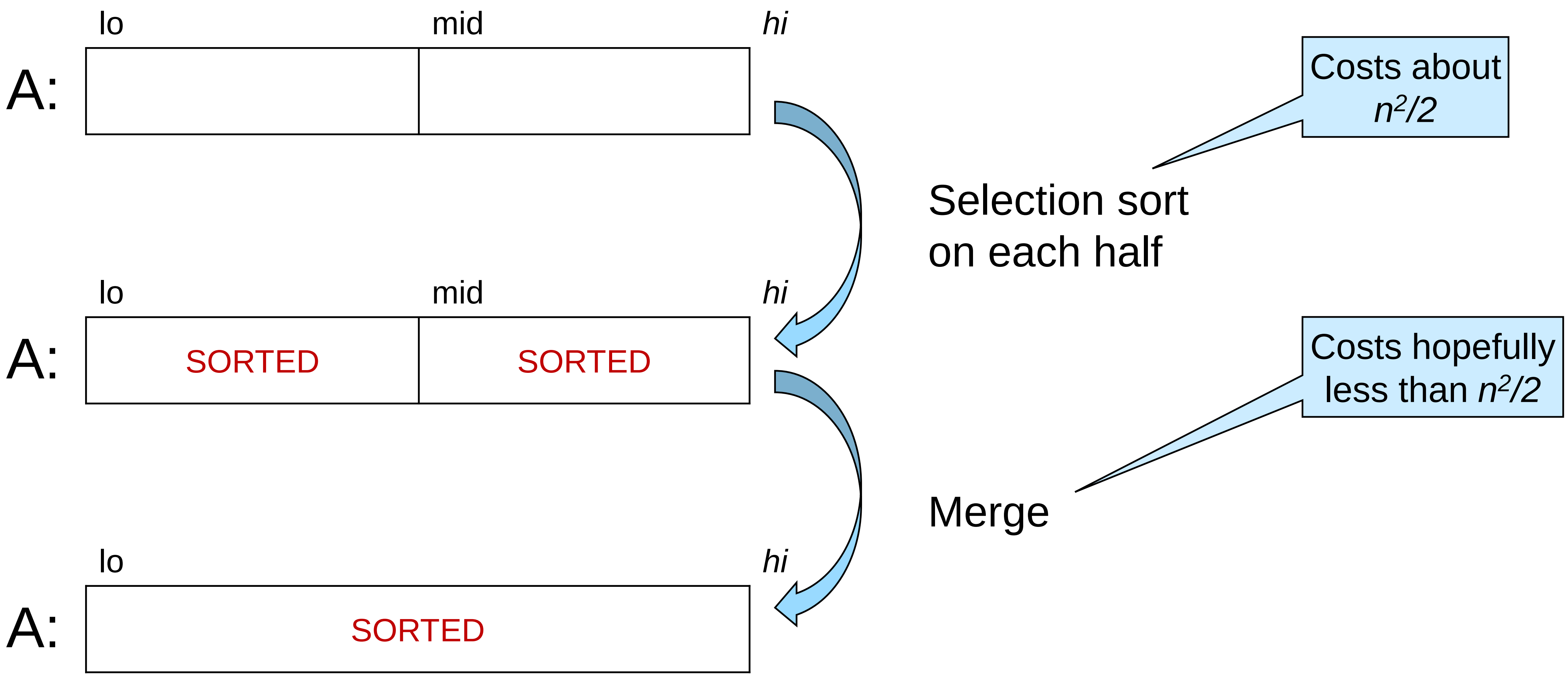
Using Selection Sort Cleverly



- ○ Sorting each half costs $n^2/4$
- ○ altogether that's $n^2/2$
- ○ that's a saving of **half** over using selection sort on the whole array!

- But the overall array is not sorted
  - ○ If we can turn two sorted halves into a sorted whole for less than $n^2/2$, we are doing better than plain selection sort

Using Selection Sort Cleverly



- merge: turns two sorted half arrays into a sorted array
  - (cheaply)

Implementation

- Computing mid

```
void sort(int[] A, int lo, int hi)
//@requires 0 <= lo && lo <= hi  && hi <= \length(A);
//@ensures is_sorted(A, lo, hi);
{
  int mid = lo + (hi - lo) / 2;
  //@assert  lo <= mid && mid <= hi;
  // … call selection sort on each half …
  // … merge the two halves …
}
```

We learned this from binary search

if hi == lo,
then mid == hi

This was not possible in the code for binary search

```
void selection_sort(int[] A, int lo, int hi)
//@requires 0 <= lo && lo <= hi  && hi <= \length(A);
//@ensures is_sorted(A, lo, hi);
```

Implementation

- Calling selection_sort on each half

```
1.  void sort(int[] A, int lo, int hi)
2.  //@requires 0 <= lo && lo <= hi  && hi <= \length(A);
3.  //@ensures is_sorted(A, lo, hi);
4.  {
5.    int mid = lo + (hi - lo) / 2;
6.    //@assert  lo <= mid && mid <= hi;
7.    selection_sort(A, lo, mid);
8.    selection_sort(A, mid, hi);
9.    // … merge the two halves
10. }
```

To show: 0 ≤ lo ≤ mid ≤ \length(A)

• 0 ≤ lo	by line 2
• lo ≤ mid	by line 6
• mid ≤ hi	by line 6
hi ≤ \length(A)	by line 2
mid ≤ \length(A)	by math

To show: 0 ≤ mid ≤ hi ≤ \length(A)
Left as exercise

- Is this code safe so far? ✓
- Since selection_sort is correct, its postcondition holds
  - A[lo, mid) sorted
  - A[mid, hi) sorted

	lo	mid	*hi*
A:	SORTED	SORTED	

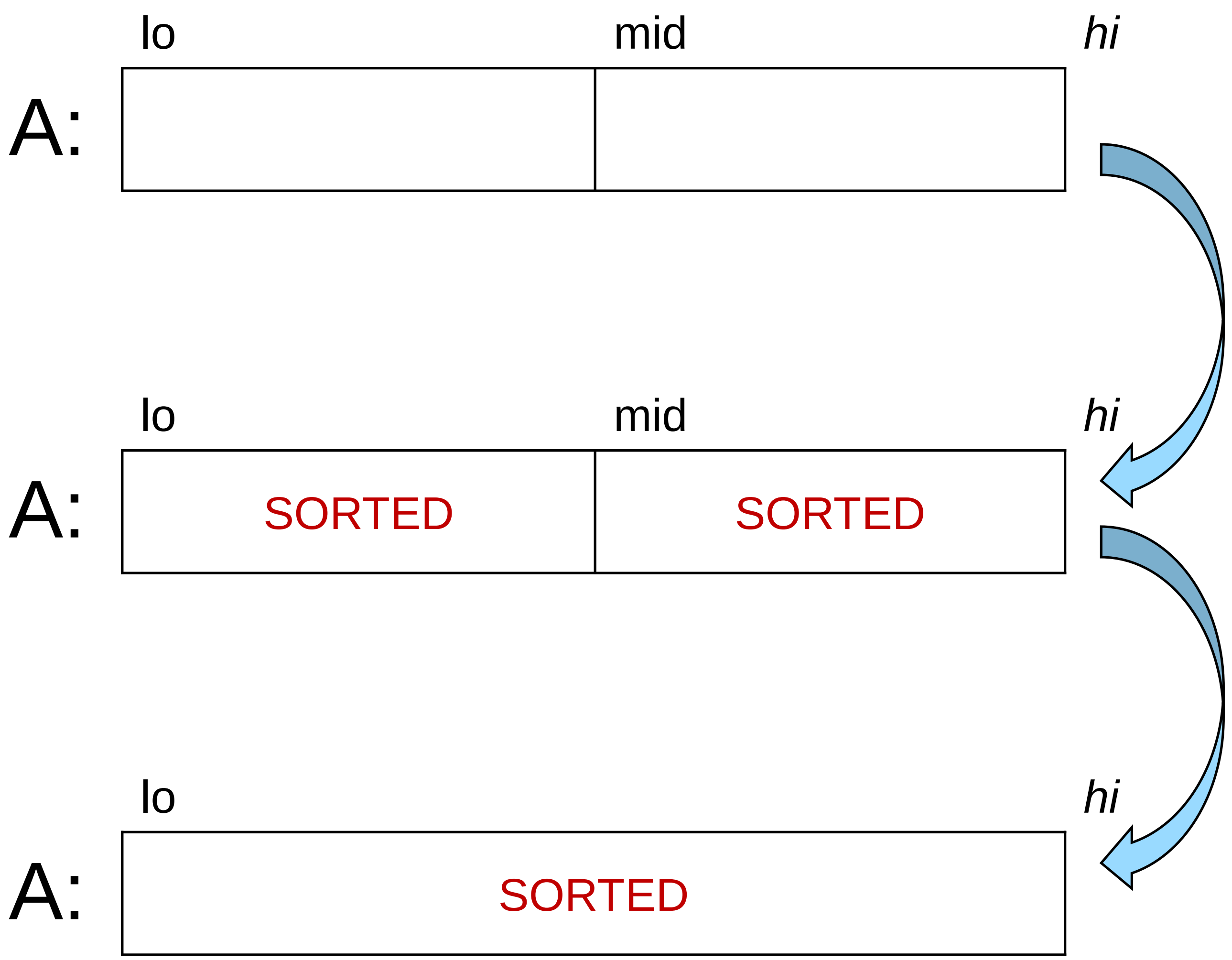
Implementation

```
void selection_sort(int[] A, int lo, int hi)
//@requires 0 <= lo && lo <= hi  && hi <= \length(A);
//@ensures is_sorted(A, lo, hi);
```

```
void sort(int[] A, int lo, int hi)
//@requires 0 <= lo && lo <= hi  && hi <= \length(A);
//@ensures is_sorted(A, lo, hi);
{
  int mid = lo + (hi - lo) / 2;
  //@assert  lo <= mid && mid <= hi;
  selection_sort(A, lo, mid);  //@assert is_sorted(A, lo, mid);
  selection_sort(A, mid, hi);  //@assert is_sorted(A, mid, hi);
  // … merge the two halves
}
```

- We are left with implementing merge

Implementation



Selection sort
on each half

Turns two
sorted half arrays segments
into a
sorted array segment

Merge

Assume we have
an implementation

```
void merge(int[] A, int lo, int mid, int hi)
//@requires 0 <= lo && lo <= mid && mid <= hi  && hi <= \length(A);
//@requires is_sorted(A, lo, mid) && is_sorted(A, mid, hi);
//@ensures is_sorted(A, lo, hi);
```

Implementation

```
void selection_sort(int[] A, int lo, int hi)
//@requires 0 <= lo && lo <= hi  && hi <= \length(A);
//@ensures is_sorted(A, lo, hi);

void merge(int[] A, int lo, int mid, int hi)
//@requires 0 <= lo && lo <= mid && mid <= hi  && hi <= \length(A);
//@requires is_sorted(A, lo, mid) && is_sorted(A, mid, hi);
//@ensures is_sorted(A, lo, hi);
```

```
void sort(int[] A, int lo, int hi)
//@requires 0 <= lo && lo <= hi  && hi <= \length(A);
//@ensures is_sorted(A, lo, hi);
{
  int mid = lo + (hi - lo) / 2;
  //@assert  lo <= mid && mid <= hi;
  selection_sort(A, lo, mid);  //@assert is_sorted(A, lo, mid);
  selection_sort(A, mid, hi);  //@assert is_sorted(A, mid, hi);
  merge(A, lo, mid, hi);
}
```

To show: 0 ≤ lo ≤ mid ≤ hi ≤ \length(A)
Left as exercise

To show: A[lo, mid) sorted **and** A[mid, hi) sorted
- by the postconditions of selection_sort

- Is this code safe? ✓
- if merge is correct, its postcondition holds
  - A[lo, hi) sorted

lo *hi*

A: SORTED

Implementation

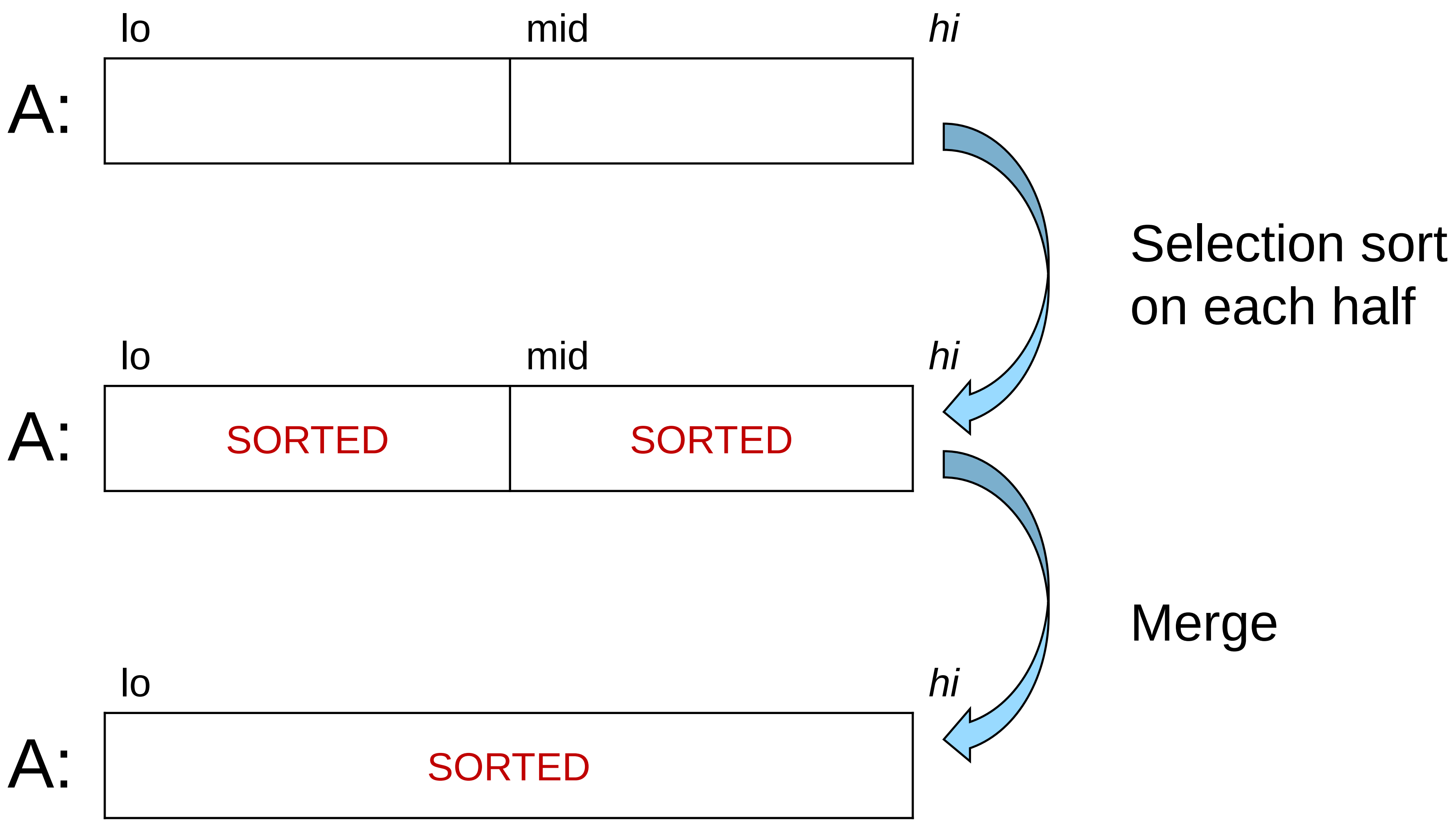
```
void selection_sort(int[] A, int lo, int hi)
//@requires 0 <= lo && lo <= hi  && hi <= \length(A);
//@ensures is_sorted(A, lo, hi);

void merge(int[] A, int lo, int mid, int hi)
//@requires 0 <= lo && lo <= mid && mid <= hi  && hi <= \length(A);
//@requires is_sorted(A, lo, mid) && is_sorted(A, mid, hi);
//@ensures is_sorted(A, lo, hi);
```

```
void sort(int[] A, int lo, int hi)
//@requires 0 <= lo && lo <= hi  && hi <= \length(A);
//@ensures is_sorted(A, lo, hi);
{
  int mid = lo + (hi - lo) / 2;
  //@assert  lo <= mid && mid <= hi;
  selection_sort(A, lo, mid);  //@assert is_sorted(A, lo, mid);
  selection_sort(A, mid, hi);  //@assert is_sorted(A, mid, hi);
  merge(A, lo, mid, hi);       //@assert is_sorted(A, lo, hi);
}
```

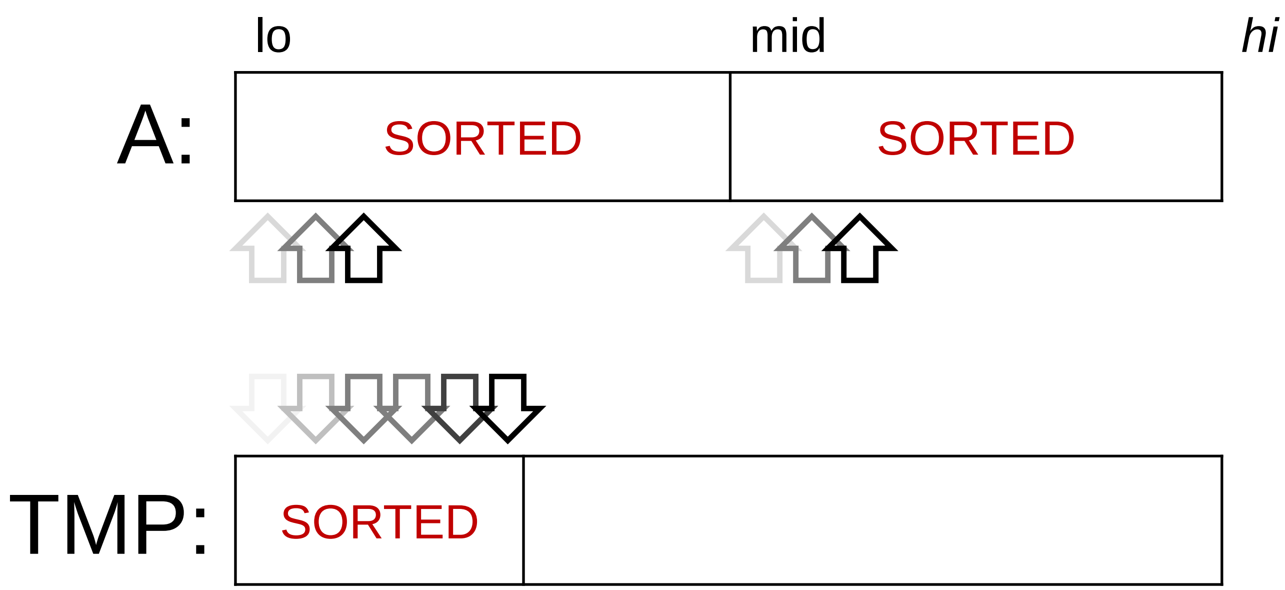
- A[lo, hi) sorted is the postcondition of sort
  - sort is **correct**

Implementation



- But how does merge work?

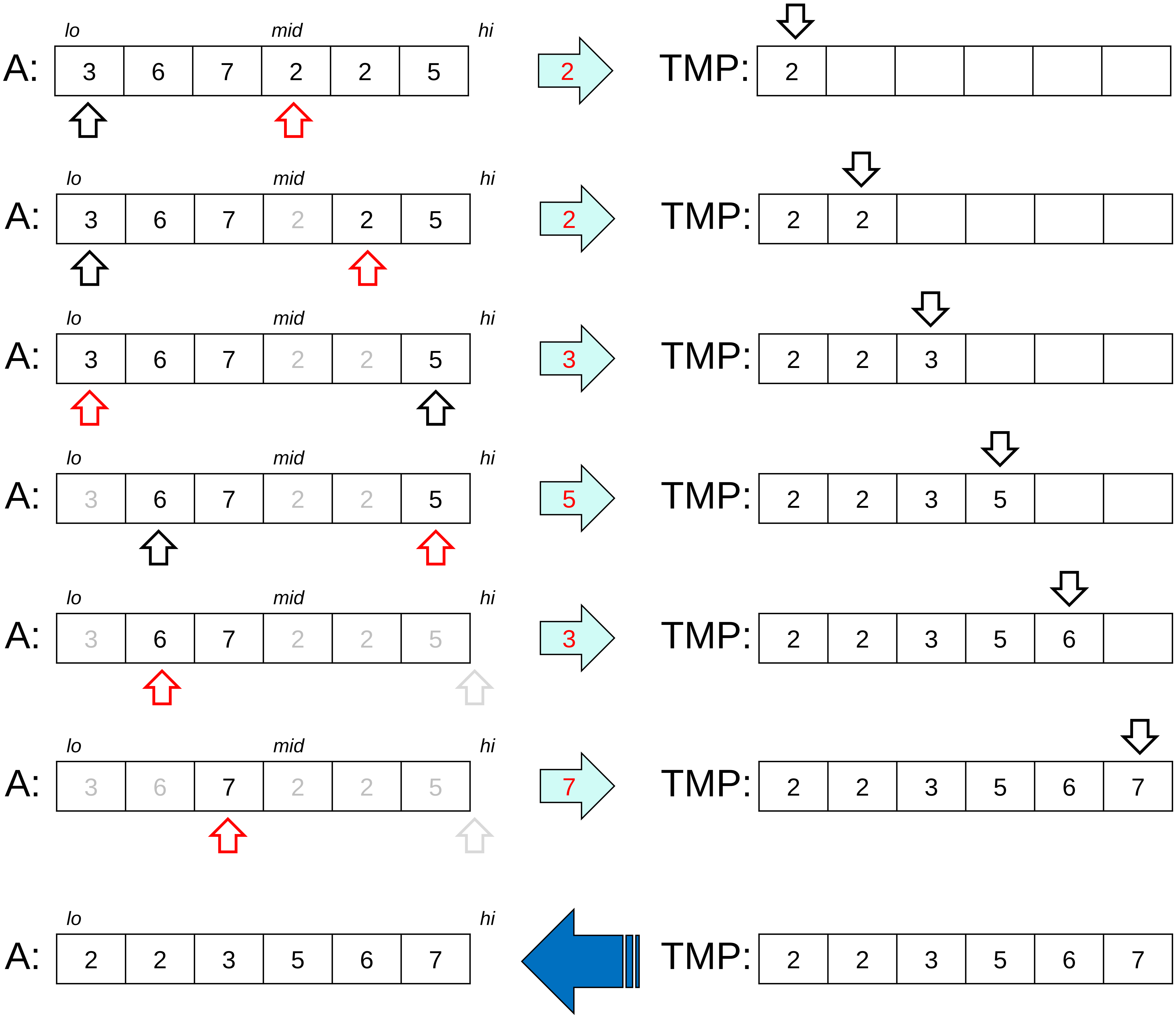
merge



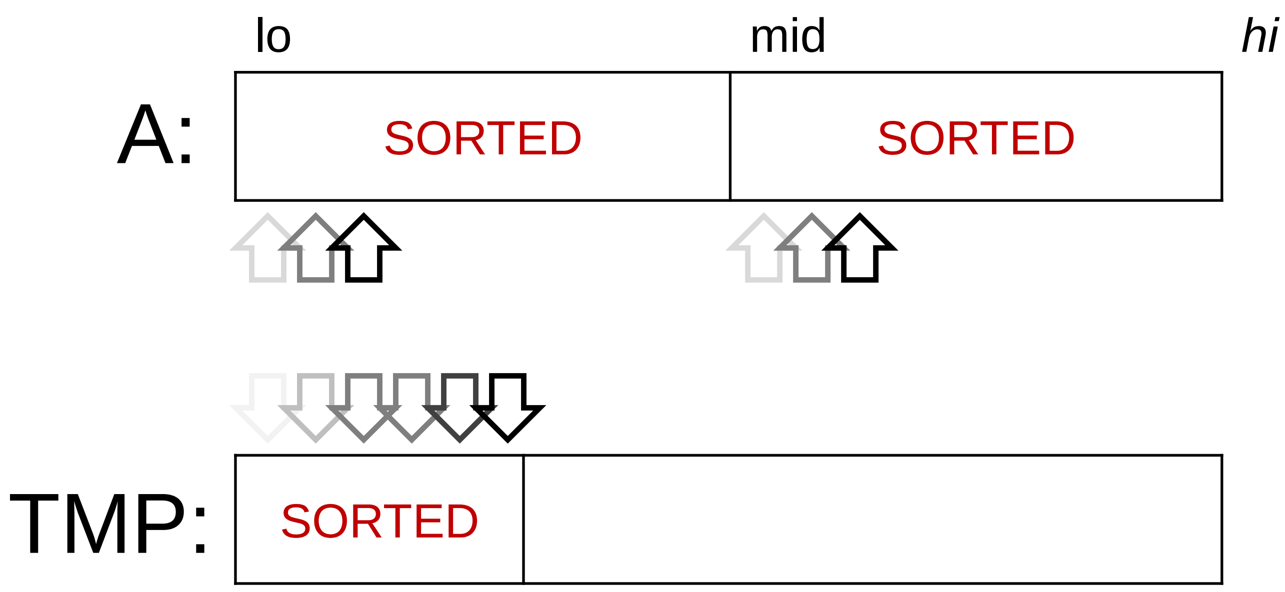
- Scan the two half array segments from left to right
- At each step, copy the smaller element in a temporary array
- Copy the temporary array back into A[lo, hi)

See code online

Example merge



merge



- Cost of merge?
  - if *A[lo, hi)* has *n* elements,
  - we copy one element to TMP at each step
    - *n* steps
  - we copy all *n* elements back to A at the end

  O(n)

- That's cheaper then $n^2/2$

In-place

- Code is **in-place** if it uses O(1) temporary storage
  - space other than its input

For example

```
void f(int[] A, int n) {
  int a = 8*n;
  bool b = false;
  char[] c = alloc_array(char, 2*n);
  string[] d = alloc_array(string, 10);
  int[] e = A;
}
```

This is a **constant amount of storage** because it takes a fixed amount of space, regardless of what n is

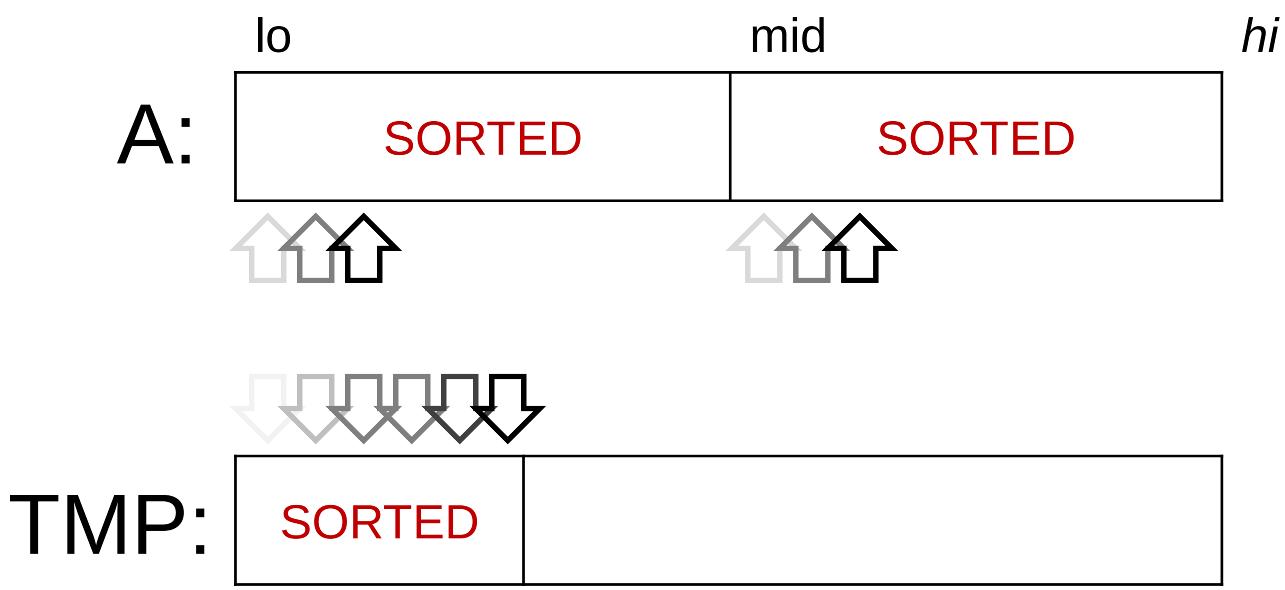
This is **not** a **constant amount of storage** because the length of c depends on the value of the parameter n

This is a **constant amount of storage** because the length of d does not depend on n

This is a **constant amount of storage** because e is just an alias to A

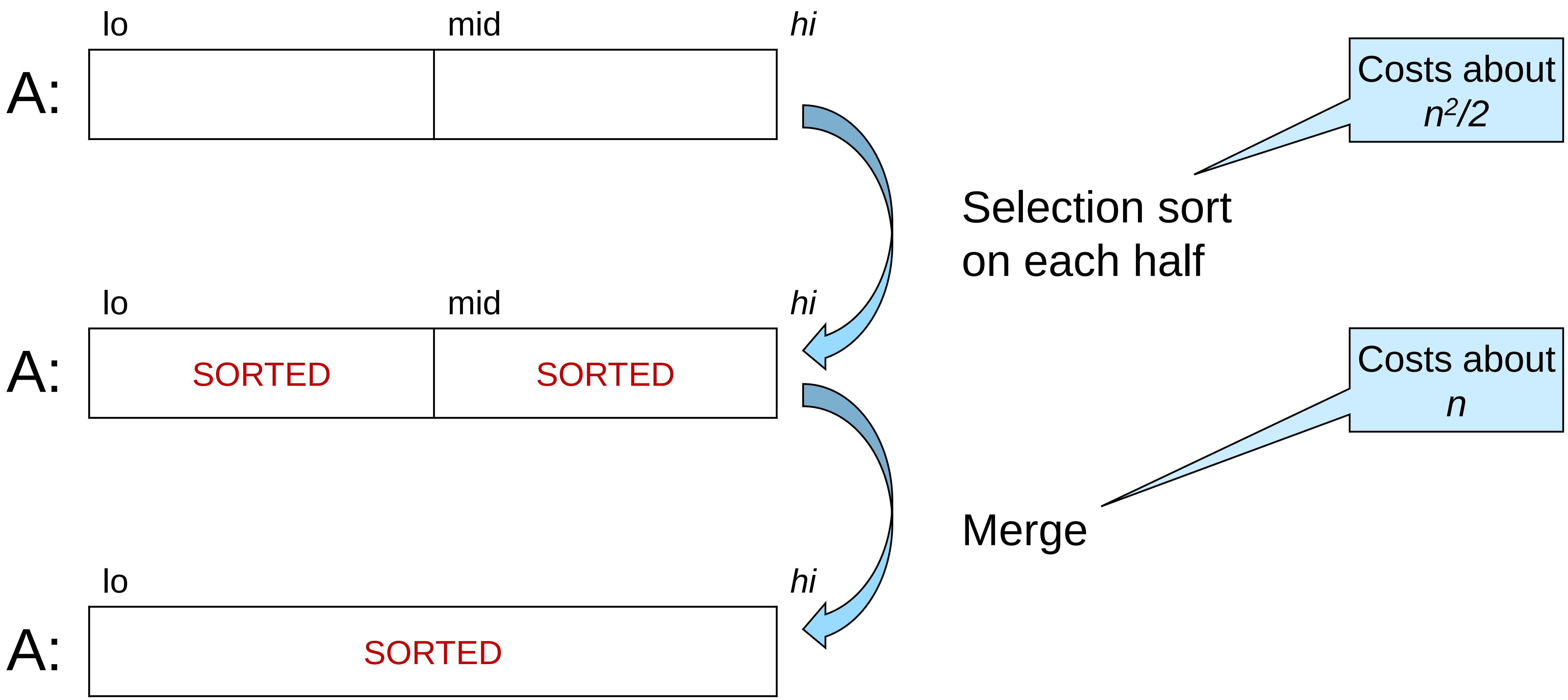
So f is not in-place

merge



- *Algorithms that use at most a constant amount of temporary storage are called **in-place***
- merge uses lots of temporary storage
  - array TMP -- same size as A[lo, hi)
  - merge is not in-place
- In-place algorithms for merge are more expensive

Using Selection Sort Cleverly



- The overall cost is about $n^2/2 + n$
  - better than plain selection sort — n^2
  - but still $O(n^2)$

Mergesort

```
void selection_sort(int[] A, int lo, int hi)
//@requires 0 <= lo && lo <= hi  && hi <= \length(A);
//@ensures is_sorted(A, lo, hi);
```

Reflection

```
void sort(int[] A, int lo, int hi)
//@requires 0 <= lo && lo <= hi  && hi <= \length(A);
//@ensures is_sorted(A, lo, hi);
{
  int mid = lo + (hi - lo) / 2;
  //@assert  lo <= mid && mid <= hi;
  selection_sort(A, lo, mid);  //@assert is_sorted(A, lo, mid);
  selection_sort(A, mid, hi);  //@assert is_sorted(A, mid, hi);
  merge(A, lo, mid, hi);       //@assert is_sorted(A, lo, hi)
}
```

- selection_sort and sort are **interchangeable**
  - they solve the **same problem** — *sorting an array segment*
  - they have the **same contracts**
  - both are **correct**

```
void selection_sort(int[] A, int lo, int hi)
//@requires 0 <= lo && lo <= hi  && hi <= \length(A);
//@ensures is_sorted(A, lo, hi);
```

A Recursive sort

```
void sort(int[] A, int lo, int hi)
//@requires 0 <= lo && lo <= hi  && hi <= \length(A);
//@ensures is_sorted(A, lo, hi);
{
  int mid = lo + (hi - lo) / 2;
  //@assert  lo <= mid && mid <= hi;
  sort(A, lo, mid);               //@assert is_sorted(A, lo, mid);
  sort(A, mid, hi);               //@assert is_sorted(A, mid, hi);
  merge(A, lo, mid, hi);          //@assert is_sorted(A, lo, hi);
}
```

- Replace the calls to selection_sort with **recursive** calls to sort
  - same preconditions: calls to sort are safe
  - same postconditions: can only produce sorted array segments
  - nothing changes for merge
    - merge produces a sorted array segment
- sort cannot compute the wrong result

A Recursive sort

```
void sort(int[] A, int lo, int hi)
//@requires 0 <= lo && lo <= hi  && hi <= \length(A);
//@ensures is_sorted(A, lo, hi);
{
  int mid = lo + (hi - lo) / 2;
  //@assert  lo <= mid && mid <= hi;
  sort(A, lo, mid);              //@assert is_sorted(A, lo, mid);
  sort(A, mid, hi);              //@assert is_sorted(A, mid, hi);
  merge(A, lo, mid, hi);         //@assert is_sorted(A, lo, hi);
}
```

- Is sort **correct**?
  - it cannot compute the wrong result
  - but will it compute the right result?
- This is a recursive function
  - but no base case!

A Recursive sort

- What if hi == lo?
  - mid == lo
  - recursive calls with identical arguments
    - infinite loop!!

- What to do?
  - A[lo,lo) is the empty array
  - always sorted!
  - simply return

```
void sort(int[] A, int lo, int hi)
//@requires 0 <= lo && lo <= hi  && hi <= \length(A);
//@ensures is_sorted(A, lo, hi);
{
  if (hi == lo) return;
  int mid = lo + (hi - lo) / 2;
  //@assert  lo <= mid && mid < hi;
  sort(A, lo, mid);              //@assert is_sorted(A, lo, mid);
  sort(A, mid, hi);              //@assert is_sorted(A, mid, hi);
  merge(A, lo, mid, hi);         //@assert is_sorted(A, lo, hi);
}
```

mid == hi now impossible

A Recursive sort

- What if hi == lo+1?
  - mid == lo, still
  - first recursive call: sort(A, lo, lo)
    - handled by the new base case
  - second recursive call: sort(A, lo, hi)
    - infinite loop!!
- What to do?
  - A[lo,lo+1) is a 1-element array
  - always sorted!
  - simply return!

```
void sort(int[] A, int lo, int hi)
//@requires 0 <= lo && lo <= hi  && hi <= \length(A);
//@ensures is_sorted(A, lo, hi);
{
  if (hi == lo) return;
  if (hi == lo+1) return;
  int mid = lo + (hi - lo) / 2;
  //@assert  lo < mid && mid < hi;
  sort(A, lo, mid);            //@assert is_sorted(A, lo, mid);
  sort(A, mid, hi);            //@assert is_sorted(A, mid, hi);
  merge(A, lo, mid, hi);       //@assert is_sorted(A, lo, hi);
}
```

mid == lo also impossible

A Recursive sort

- No more opportunities for infinite loops
- The preconditions still imply the postconditions
  - base case return: arrays of lengths 0 and 1 are always sorted
  - final return: our original proof applies

- sort is **correct!**

- This function is called

  mergesort

```
void sort(int[] A, int lo, int hi)
//@requires 0 <= lo && lo <= hi  && hi <= \length(A);
//@ensures is_sorted(A, lo, hi);
{
  if (hi - lo <= 1) return;
  int mid = lo + (hi - lo) / 2;
  //@assert  lo <= mid && mid <= hi;
  sort(A, lo, mid);               //@assert is_sorted(A, lo, mid);
  sort(A, mid, hi);               //@assert is_sorted(A, mid, hi);
  merge(A, lo, mid, hi);          //@assert is_sorted(A, lo, hi);
}
```

minor clean-up

A Recursive sort

- Recursive functions don't have loop invariants
- How does our correctness methodology transfer?
  - **INIT**: Safety of the initial call to the function
  - **PRES**: From the preconditions to the safety of the recursive calls
  - **EXIT**: From the postconditions of the recursive calls to the postcondition of the function
  - **TERM**:
    - the base case handles input smaller than some bound
    - the input of each recursive call is strictly smaller than the input of the function

Mergesort

```
void merge(int[] A, int lo, int mid, int hi)
//@requires 0 <= lo && lo <= mid && mid <= hi  && hi <= \length(A);
//@requires is_sorted(A, lo, mid) && is_sorted(A, mid, hi);
//@ensures is_sorted(A, lo, hi);
```

```
void mergesort(int[] A, int lo, int hi)
//@requires 0 <= lo && lo <= hi  && hi <= \length(A);
//@ensures is_sorted(A, lo, hi);
{
  if (hi - lo <= 1) return;

  int mid = lo + (hi - lo) / 2;
  //@assert  lo < mid && mid < hi;
  mergesort(A, lo, mid);
  mergesort(A, mid, hi);
  merge(A, lo, mid, hi);
}
```

Complexity of Mergesort

```
void mergesort(int[] A, int lo, int hi) {
  if (hi - lo <= 1) return;       // O(1)
  int mid = lo + (hi - lo) / 2;  // O(1)
  mergesort(A, lo, mid);
  mergesort(A, mid, hi);
  merge(A, lo, mid, hi);        // O(n)
}
```

n = hi - lo

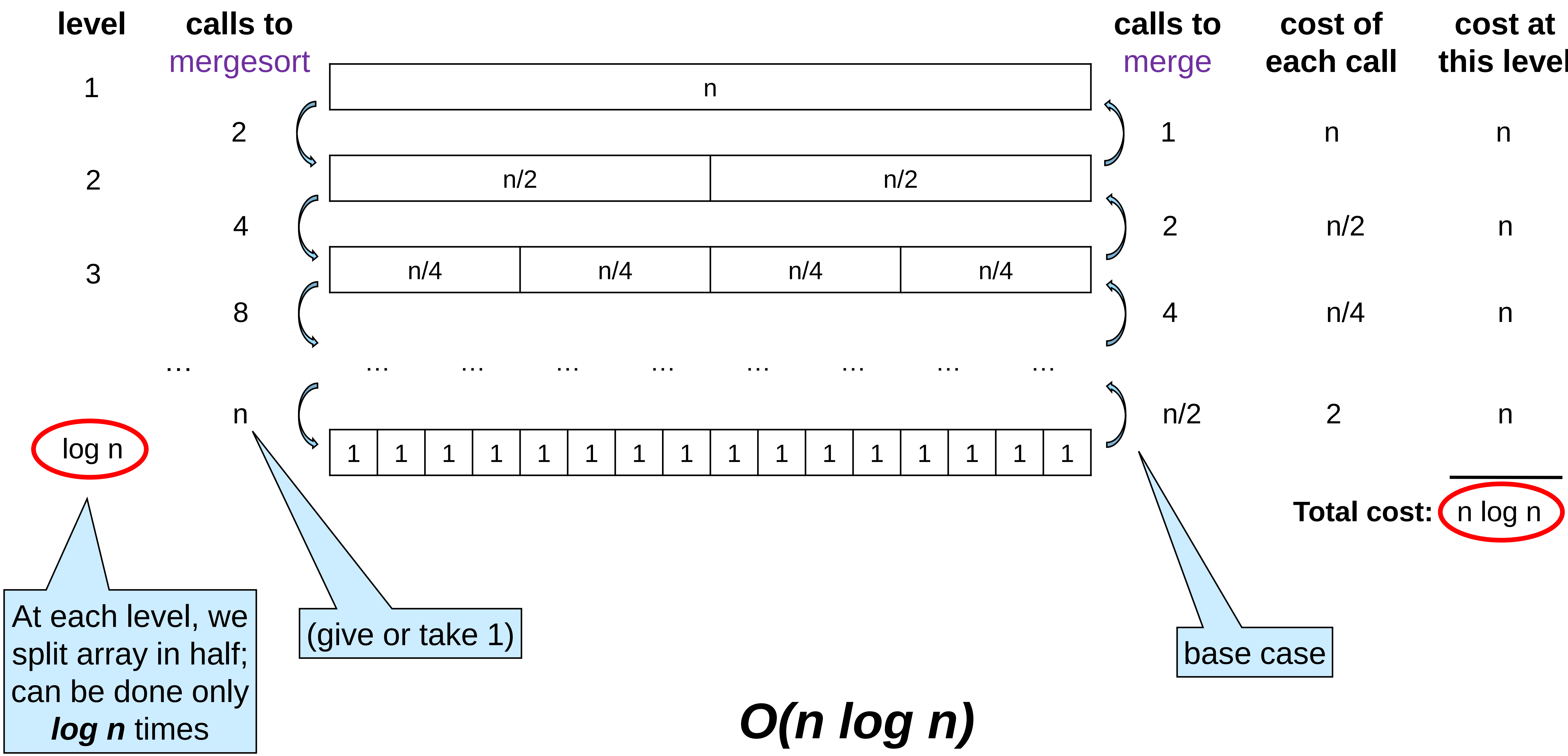
- Work done by each call to mergesort

  (ignoring recursive calls)
  - Base case: constant cost -- O(1)
  - Recursive case:
    - compute mid: constant cost -- O(1)
    - recursive calls: (ignored)
    - merge: linear cost -- O(n)

- We need to add this for all recursive calls
  - It is convenient to organize them by *level*

Complexity of Mergesort

```
void mergesort(int[] A, int lo, int hi) {
  if (hi - lo <= 1) return;        // O(1)
  int mid = lo + (hi - lo) / 2;  // O(1)
  mergesort(A, lo, mid);
  mergesort(A, mid, hi);
  merge(A, lo, mid, hi);          // O(n)
}
```



O(n log n)

Comparing Sorting Algorithms

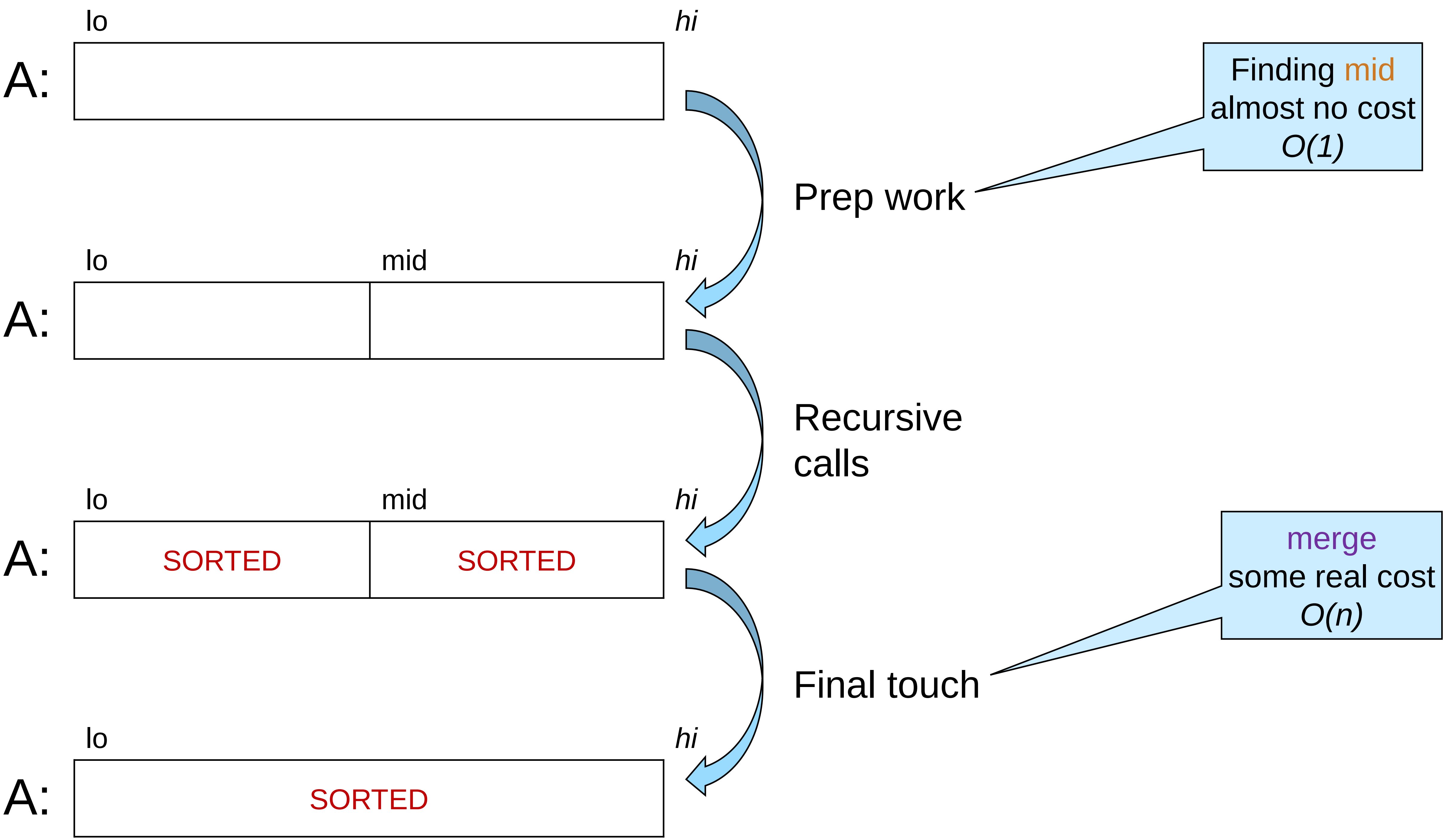
- Selection sort and mergesort solve the **same problem**
  - mergesort is **asymptotically faster**: $O(n \log n)$ vs. $O(n^2)$
    - mergesort is preferable if speed for large inputs is all that matters
  - selection sort is **in-place** but mergesort is not
    - selection sort may be preferable if space is very tight
- Choosing an algorithm involves several parameters
  - **It depends on the application**
- Summary

	Selection sort	**Mergesort**
Worst-case complexity	$O(n^2)$	$O(n \log n)$
In-place?	Yes	No

Quicksort

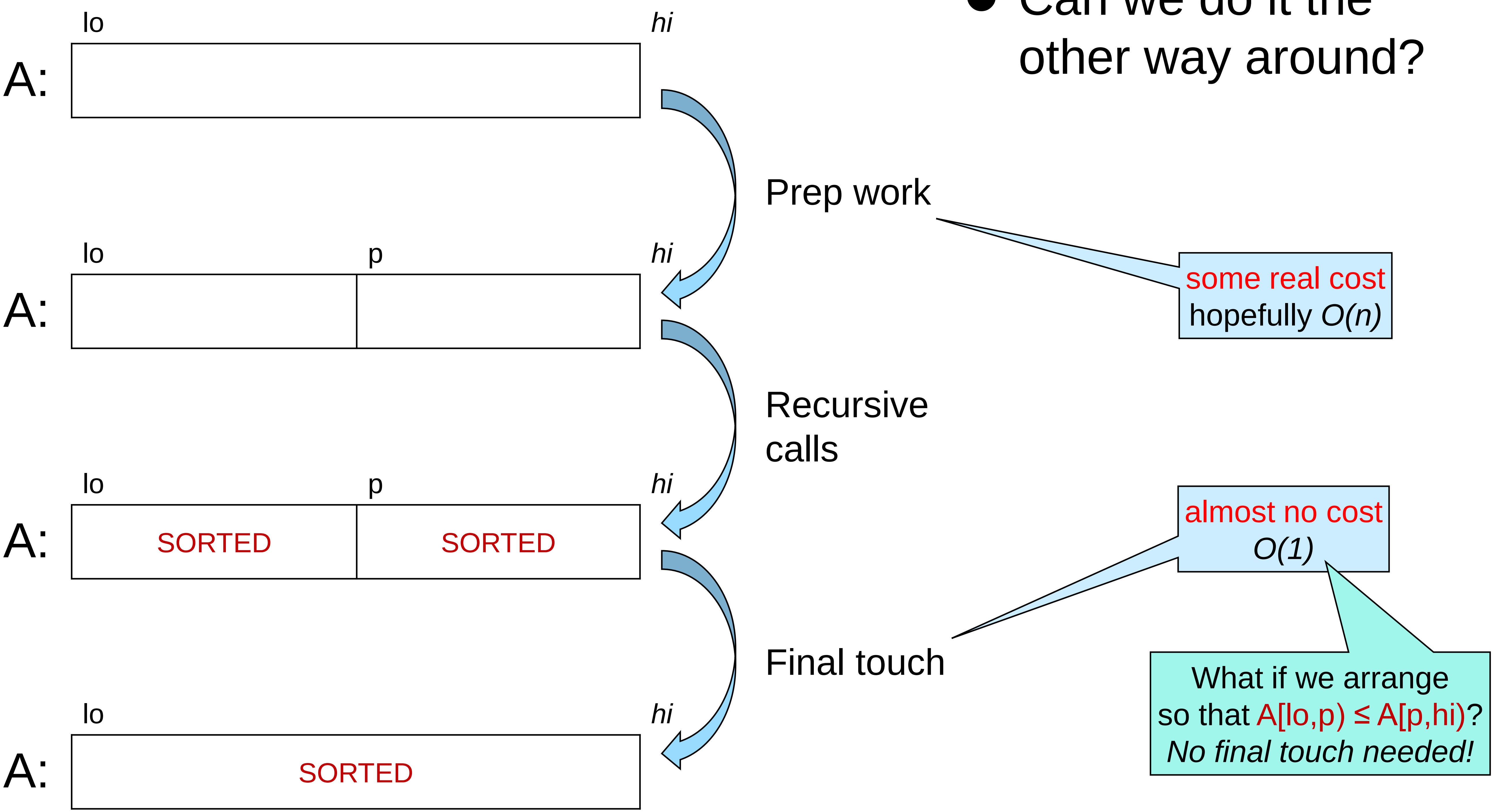
Reflections

```
void mergesort(int[] A, int lo, int hi) {
  if (hi - lo <= 1) return;
  int mid = lo + (hi - lo) / 2;
  mergesort(A, lo, mid);
  mergesort(A, mid, hi);
  merge(A, lo, mid, hi);
}
```



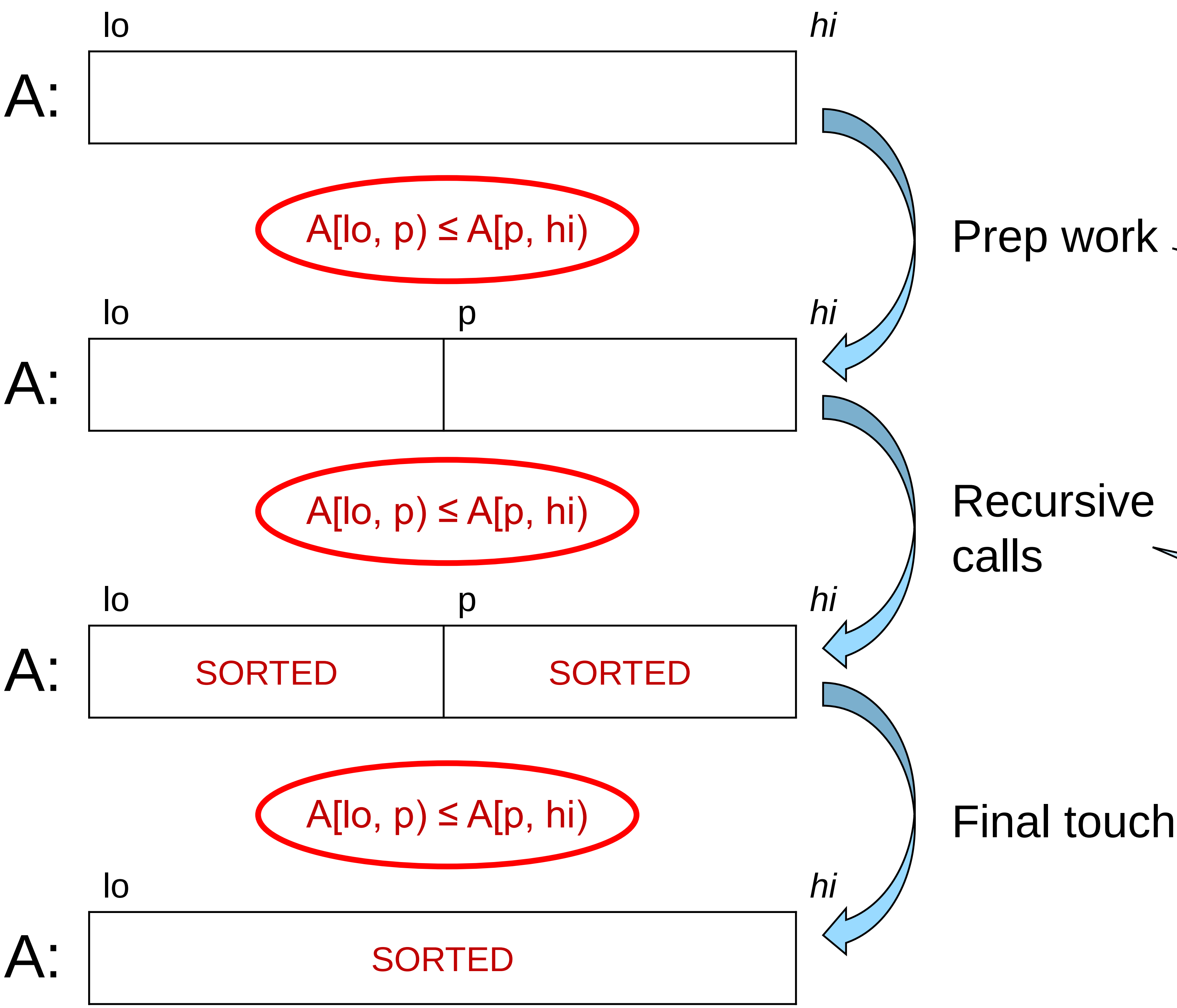
Reflections

- Can we do it the other way around?



Reflections

- ***How*** do we do it the other way around?



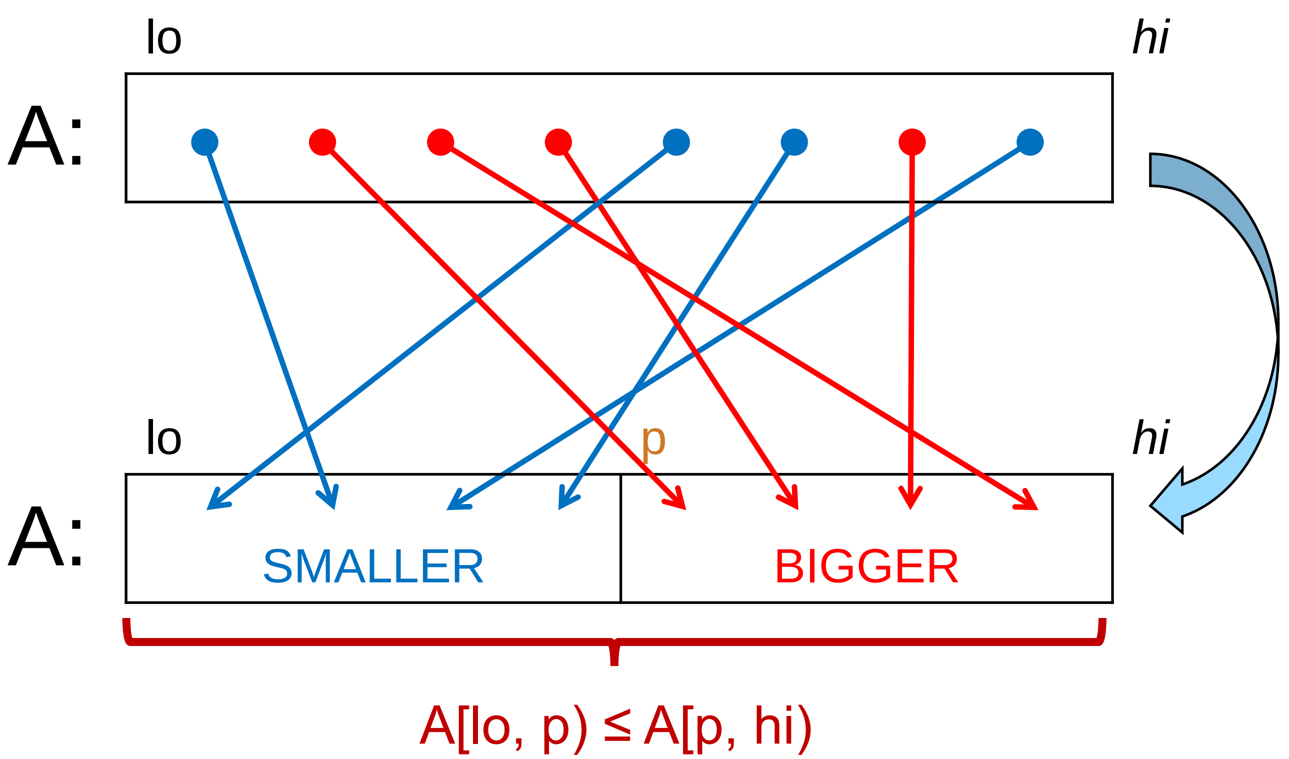
some real cost
hopefully *O(n)*

Applied independently
on each section:
if A[lo,p) ≤ A[p,hi) *before*,
then A[lo,p) ≤ A[p,hi) *after*

Nothing to do

Partition

- A function that
  - moves small values to the left of A
  - moves big values to the right of A
  - returns the index p that separates them



- This is partition

```
int partition (int[] A, int lo, int hi)
//@requires 0 <= lo && lo <= hi  && hi <= \length(A);
//@ensures lo <= \result && \result <= hi;
//@ensures le_segs(A, lo, \result, A, \result, hi);
```

Partition

- Using partition in sort

```
int partition (int[] A, int lo, int hi)
//@requires 0 <= lo && lo <= hi  && hi <= \length(A);
//@ensures lo <= \result && \result <= hi;
//@ensures le_segs(A, lo, \result, A, \result, hi);
```

```
void sort(int[] A, int lo, int hi)
//@requires 0 <= lo && lo <= hi  && hi <= \length(A);
//@ensures is_sorted(A, lo, hi);
{
  if (hi - lo <= 1) return;

  int p = partition(A, lo, hi);
  //@assert  lo <= p && p <= hi;
  sort(A, lo, p);
  sort(A, p, hi);
}
```

just like mergesort

- What if p == hi where hi > lo+1 ?
  - Infinite loop!

- We want p < hi
- Thus \result < hi in partition

Partition

- To use partition in sort, we need \result < hi

```
int partition (int[] A, int lo, int hi)
//@requires 0 <= lo && lo <= hi  && hi <= \length(A);
//@ensures lo <= \result && \result < hi;
//@ensures le_segs(A, lo, \result, A, \result, hi);
```

DANGER!
this function is **unimplementable**!
if hi==lo,
then \result can't exist

- We want lo < hi

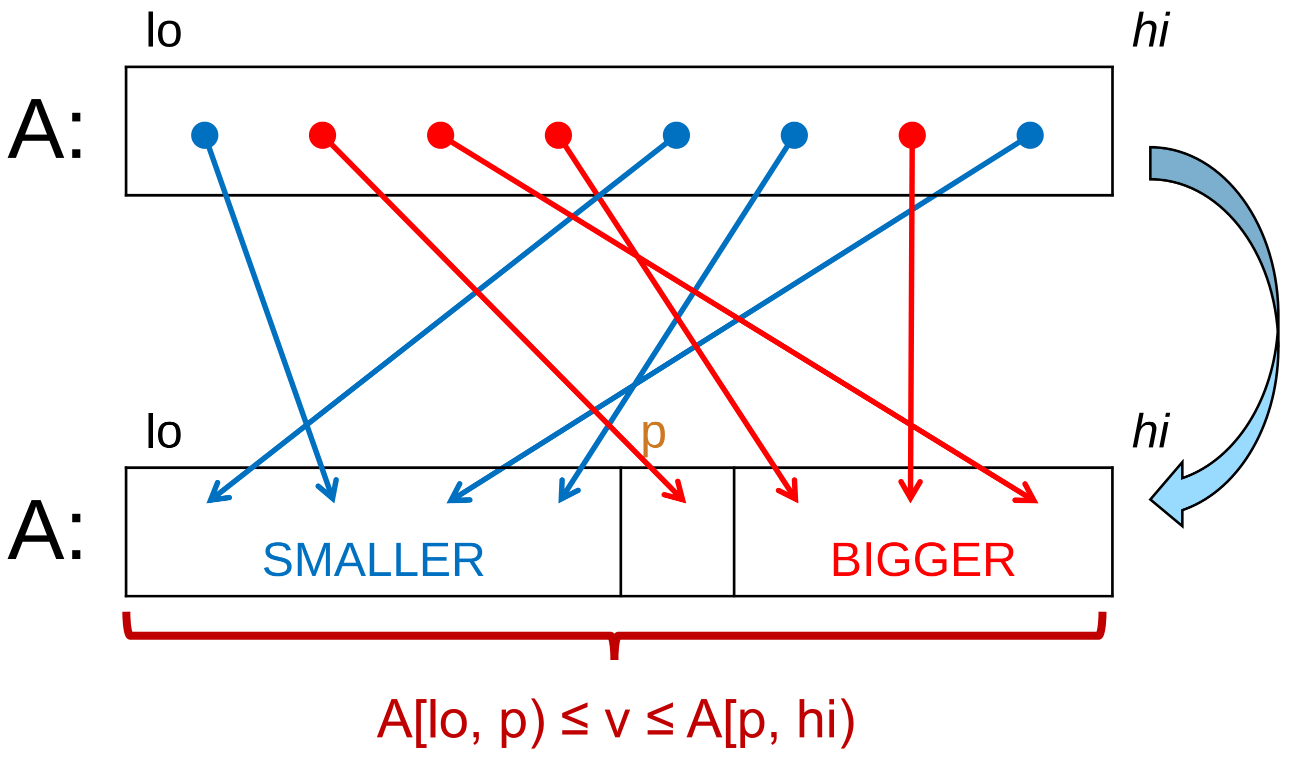
```
int partition (int[] A, int lo, int hi)
//@requires 0 <= lo && lo < hi  && hi <= \length(A);
//@ensures lo <= \result && \result < hi;
//@ensures le_segs(A, lo, \result, A, \result, hi);
```

  - There is an element at A[\result]

Partition

- If A[lo, p) ≤ A[p, hi), then there is a value v such that

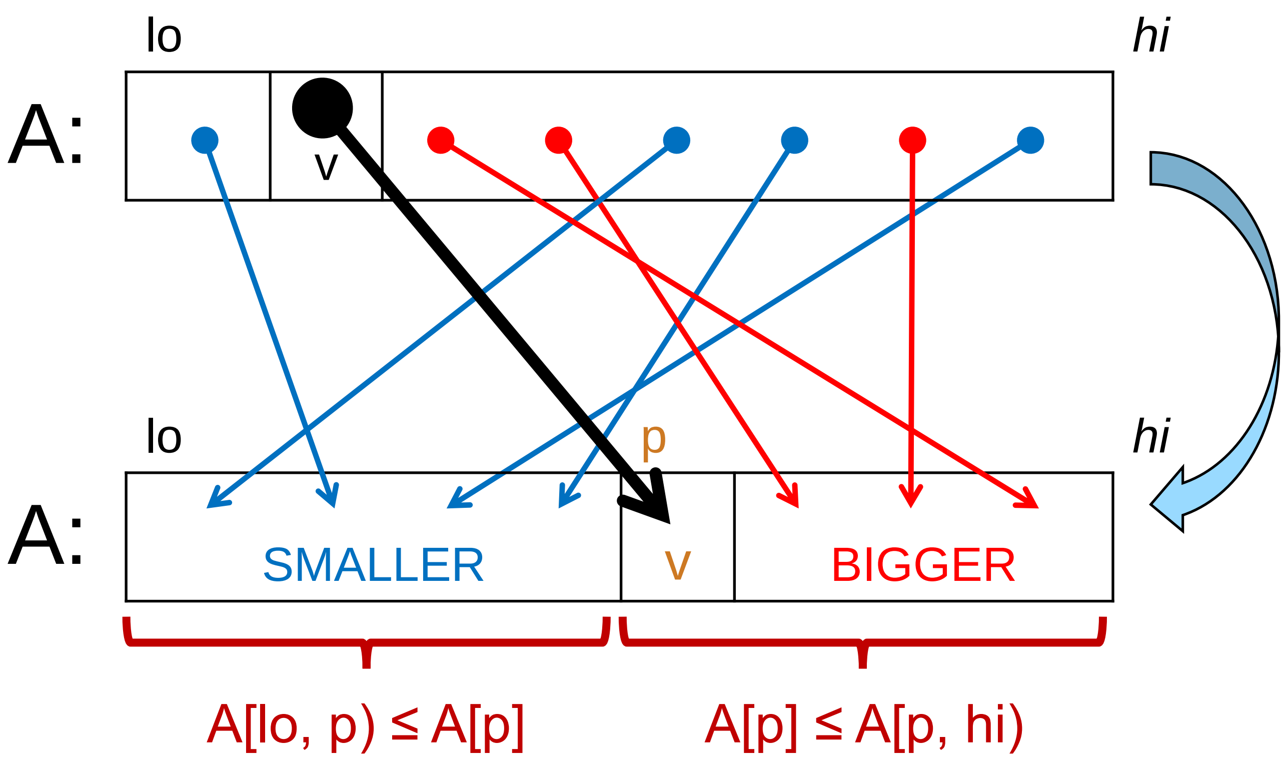
A[lo, p) ≤ v ≤ A[p, hi)



Partition

- *There is a value v s.t. A[lo, p) ≤ v ≤ A[p, hi)*

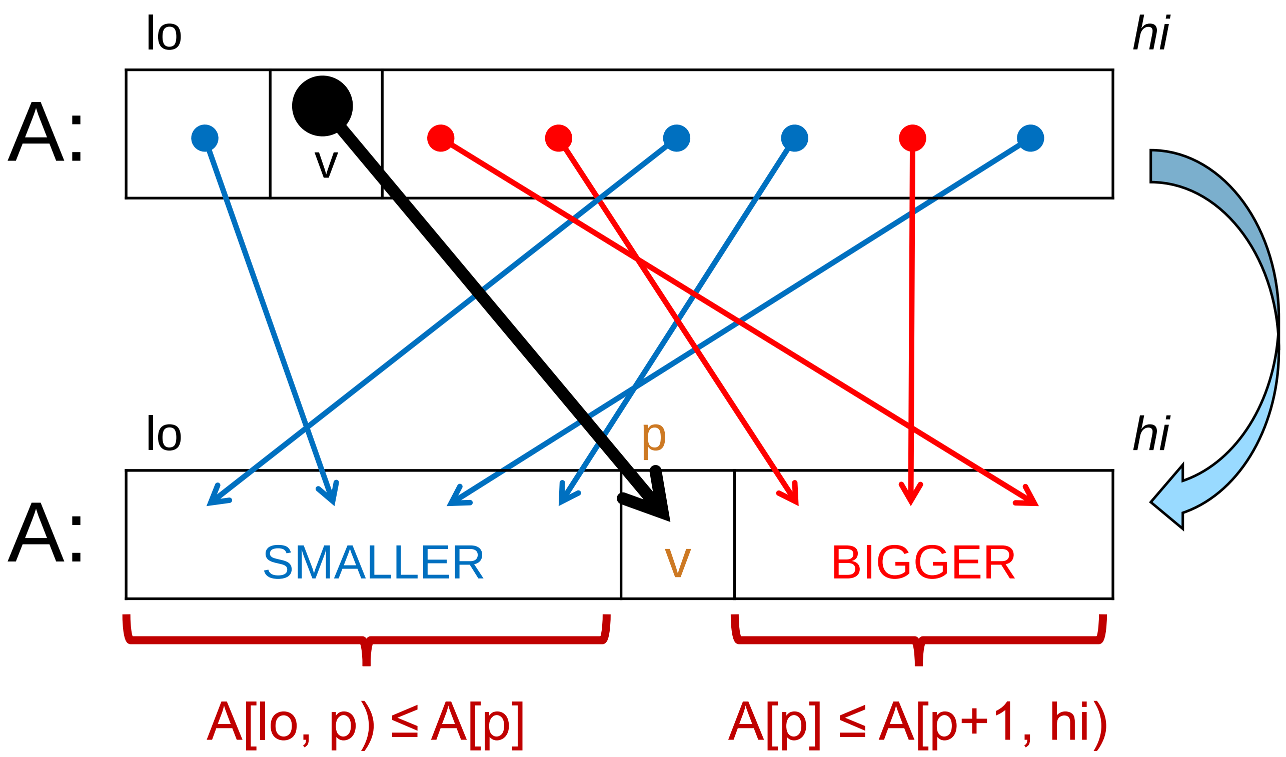
- Since lo < hi, take v to be an element of A[lo, hi)
  - It will end up in A[p]
  - A[lo, p) ≤ A[p] ≤ A[p, hi)



- v is called the **pivot**
  - p is the **pivot index**

Partition

- $A[lo, p) \leq A[p] \leq A[p, hi)$
  is equivalent to
  - $A[lo, p) \leq A[p]$
  - $A[p] \leq A[p+1, hi)$




```
int partition (int[] A, int lo, int hi)
//@requires 0 <= lo && lo < hi  && hi <= \length(A);
//@ensures lo <= \result && \result < hi;
//@ensures ge_seg(A[\result], A, lo, \result);
//@ensures le_seg(A[\result], A, \result+1, hi);
```

- the pivot separates the smaller and the bigger elements
- it ends up in the right place in the sorted array

Quicksort

```
int partition (int[] A, int lo, int hi)
//@requires 0 <= lo && lo < hi  && hi <= \length(A);
//@ensures lo <= \result && \result < hi;
//@ensures ge_seg(A[\result], A, lo, \result);
//@ensures le_seg(A[\result], A, \result+1, hi);
```

- This algorithm is called **quicksort**

```
void quicksort(int[] A, int lo, int hi)
//@requires 0 <= lo && lo <= hi  && hi <= \length(A);
//@ensures is_sorted(A, lo, hi);
{
  if (hi - lo <= 1) return;

  int p = partition(A, lo, hi);
  //@assert  lo <= p && p < hi;
  quicksort(A, lo, p);
  quicksort(A, p+1, hi);
}
```

pivot A[p] is already in the right place

Quicksort

```
int partition (int[] A, int lo, int hi)
//@requires 0 <= lo && lo < hi  && hi <= \length(A);
//@ensures lo <= \result && \result < hi;
//@ensures ge_seg(A[\result], A, lo, \result);
//@ensures le_seg(A[\result], A, \result+1, hi);
```

- Is it safe?

```
1. void quicksort(int[] A, int lo, int hi)
2. //@requires 0 <= lo && lo <= hi  && hi <= \length(A);
3. //@ensures is_sorted(A, lo, hi);
4. {
5.   if (hi - lo <= 1) return;

6.   int p = partition(A, lo, hi);
7.   //@assert  lo <= p && p < hi;
8.   quicksort(A, lo, p);
9.   quicksort(A, p+1, hi);
10.}
```

To show: $0 \leq lo < hi \leq \backslash length(A)$

- $0 \leq lo$ by line 2
- $lo \leq hi+1$ by line 5
  $lo < hi$ by math
- $hi \leq \backslash length(A)$ by line 2

To show: $0 \leq lo \leq p \leq \backslash length(A)$
Like mergesort

To show: $0 \leq p+1 \leq hi \leq \backslash length(A)$
Left as exercise

✓

Quicksort

```
int partition (int[] A, int lo, int hi)
//@requires 0 <= lo && lo < hi  && hi <= \length(A);
//@ensures lo <= \result && \result < hi;
//@ensures ge_seg(A[\result], A, lo, \result);
//@ensures le_seg(A[\result], A, \result+1, hi);
```

- Is it correct?

```
1.  void quicksort(int[] A, int lo, int hi)
2.  //@requires 0 <= lo && lo <= hi  && hi <= \length(A);
3.  //@ensures is_sorted(A, lo, hi);
4.  {
5.    if (hi - lo <= 1) return;
6.    int p = partition(A, lo, hi);
7.    //@assert  lo <= p && p < hi;
8.    //@assert ge_seg(A[p], A, lo, p);
9.    //@assert le_seg(A[p], A, p+1, hi);
10.   quicksort(A, lo, p);     //@assert is_sorted(A, lo, p);
11.   quicksort(A, p+1, hi); //@assert is_sorted(A, p+1, hi);
12. }
```

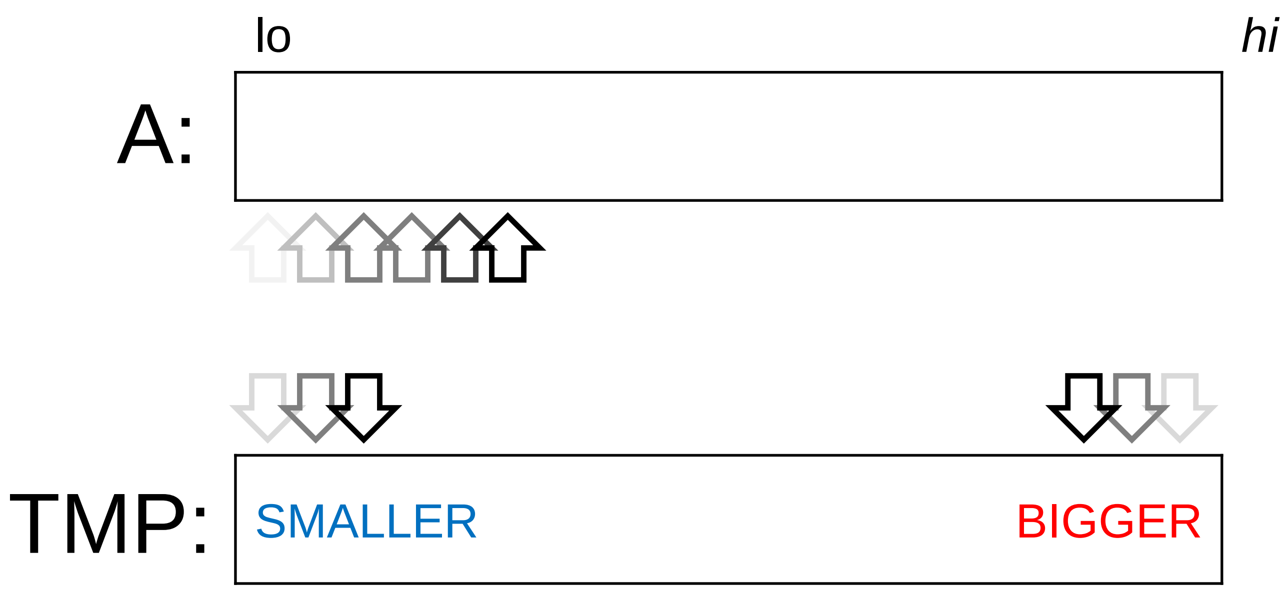
To show: A[lo, hi) sorted
All arrays of length 0 or 1
are sorted

To show: A[lo, hi) sorted

A.	A[lo, p) ≤ A[p]	by line 8
B.	A[p] ≤ A[p+1, hi)	by line 9
C.	A[lo, p) sorted	by line10
D.	A[p+1, hi) sorted	by line11
E.	A[lo, hi) sorted	by A-D

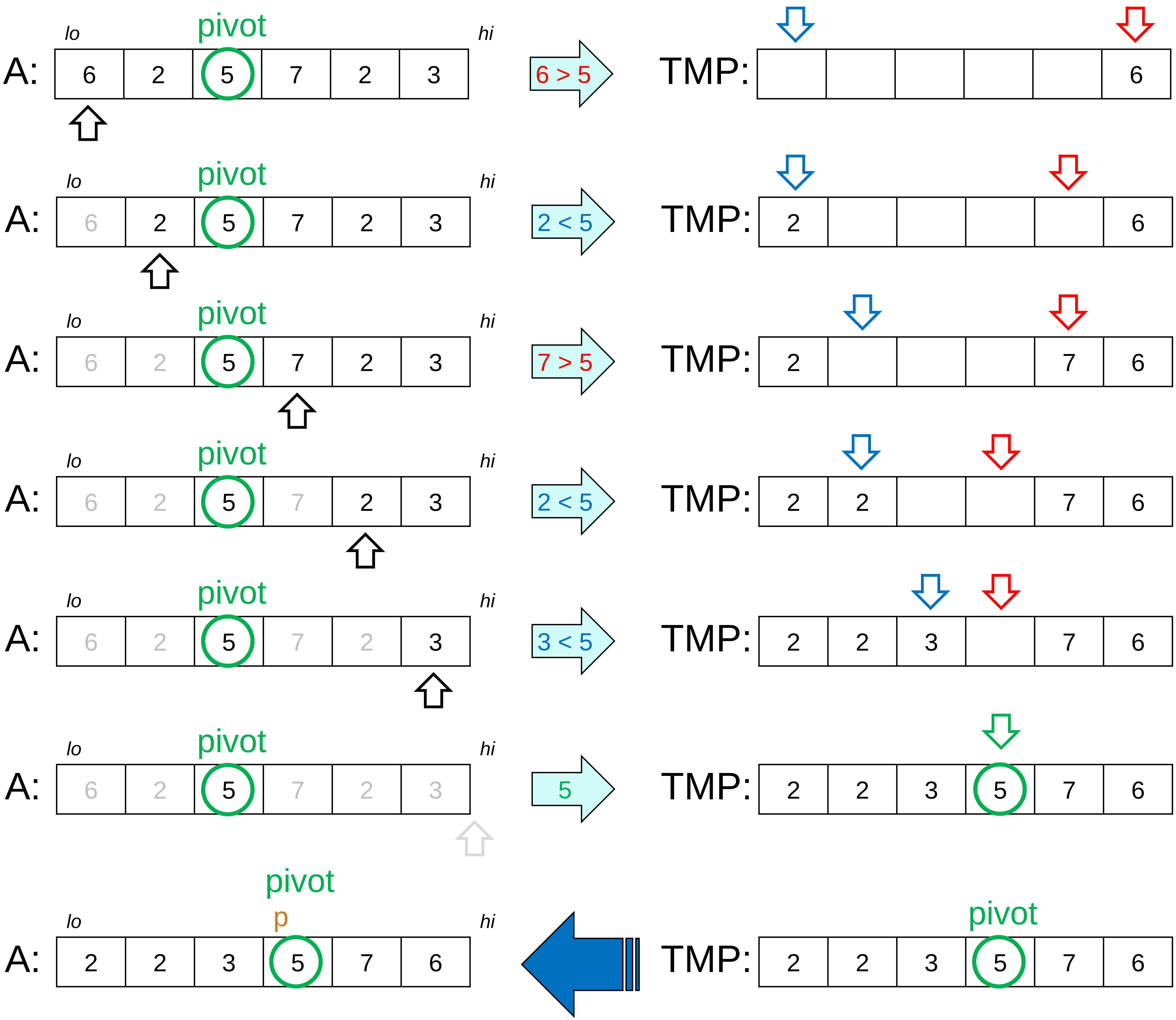
✓

How to partition

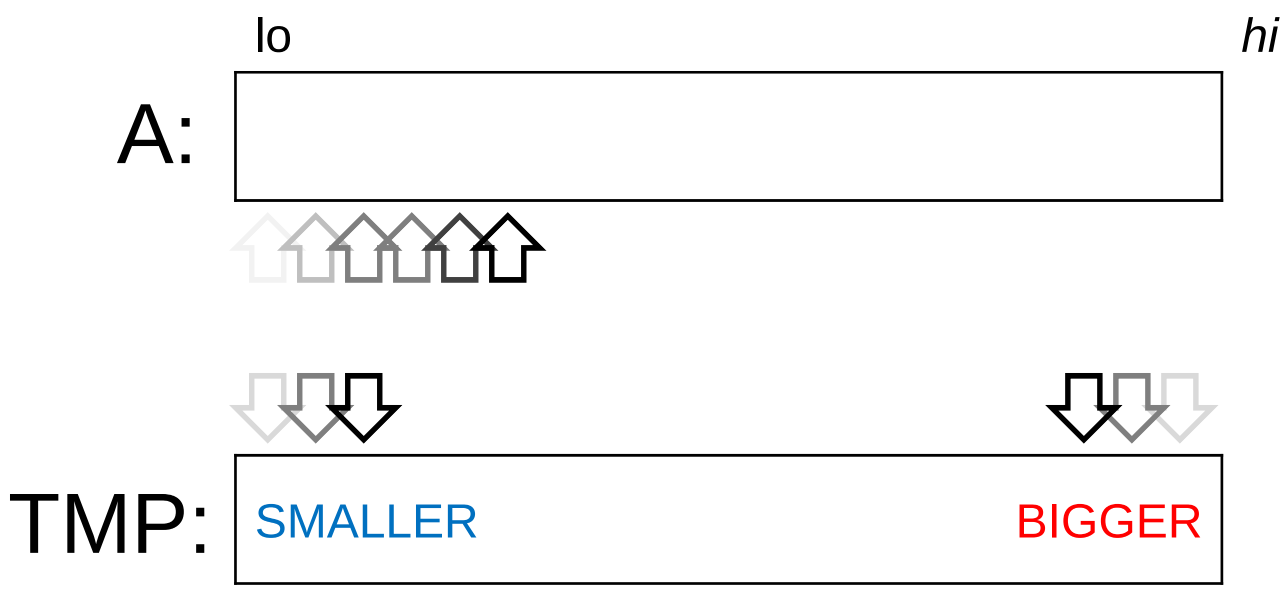


- Create a temporary array, TMP, the same size as A[lo, hi)
- Pick the pivot in the array
- Put all other elements at either end of TMP
  - smaller on the left, larger on the right
- Put pivot in the one spot left
- Copy TMP back into A[lo, hi)
- Return the index where the pivot ends up

Example partition



How to partition

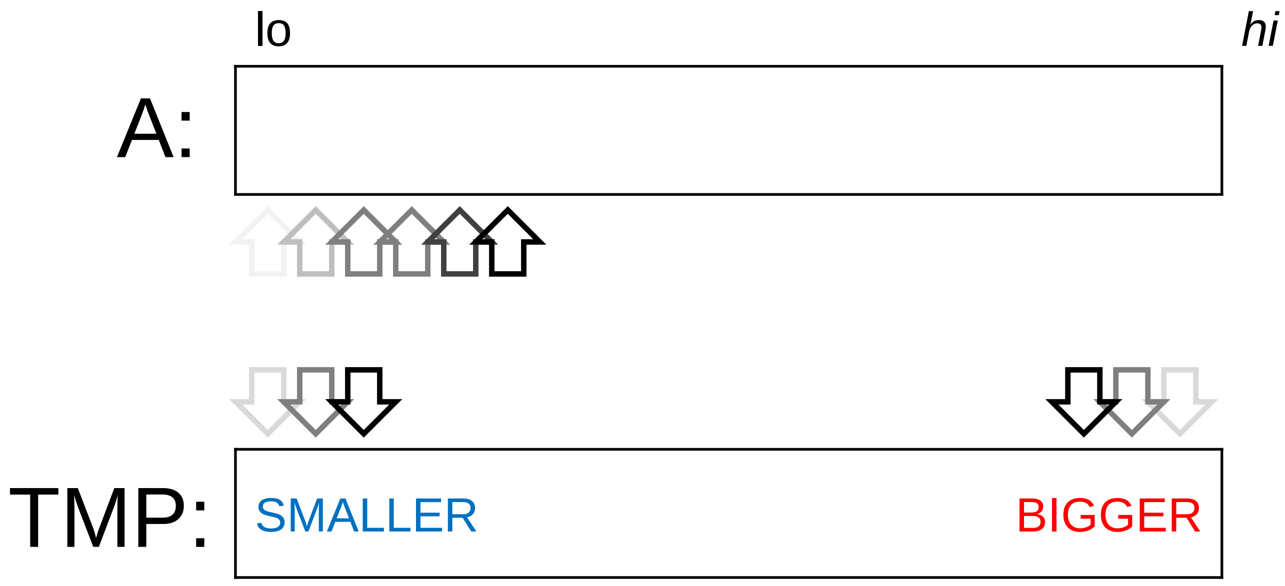


- Cost of partition?
  - if *A[lo, hi)* has *n* elements,
  - we copy one element to TMP at each step
    - *n* steps
  - we copy all *n* elements back to A at the end

O(n)

- Just like merge

How to partition



- Done this way, partition is **not** in-place
- With a little cleverness, this can be modified to be **in-place**
  - Still ***O(n)***

See code online

Complexity of Quicksort

```
void quicksort(int[] A, int lo, int hi) {
  if (hi - lo <= 1) return;         // O(1)
  int p = partition(A, lo, hi);   // O(n)
  quicksort(A, lo, p);
  quicksort(A, p+1, hi);
}
```

- If we pick the **median** of A[lo, hi) as the pivot,
  - the median is the value such that half elements are larger and half smaller
  - the pivot index then becomes the **midpoint**, *(lo + hi)/2*

then it's like mergesort

level	**calls to** quicksort		**calls to** partition	**cost of each call**	**cost at this level**
1		n			
	2		1	n	n
2		n/2 | n/2			
	4		2	n/2	n
3		n/4 | n/4 | n/4 | n/4			
	8		4	n/4	n
…		… … … … … … … …			
	n		n/2	2	n
log n		1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1			

Total cost: n log n

At each level, we split array in half; can be done only ***log n*** times

(give or take 1)

base case

O(n log n)

Complexity of Quicksort

- How do we find the median?
  - sort the array and pick the element at the midpoint …
  - This defeats the purpose!
  - And it costs *O(n log n)* -- using mergesort

- We want to spent at most *O(n)*
- No such algorithm for finding the median!
  - Either *O(n log n)*
  - Or *O(n)* for an approximate solution
    - which may be an Ok compromise

- So, ***if we are lucky***, quicksort has cost O(n log n)

Complexity of Quicksort

```
void quicksort(int[] A, int lo, int hi) {
  if (hi - lo <= 1) return;        // O(1)
  int p = partition(A, lo, hi);   // O(n)
  quicksort(A, lo, p);
  quicksort(A, p+1, hi);
}
```

- What if we are **unlucky**?
  - Pick the **smallest** element each time *(or the largest)*

level	calls to quicksort		calls to partition	cost of each call	cost at this level
1		n			
	2		1	n	n
2		n-1			
	2		1	n-1	n-1
3		n-2			
	2		1	n-2	n-2
…		… … … … … … … …			
	2		1	1	1
n		1			

Total cost: n(n+1)/2

(give or take 1)

base case

At level *i*, we make one recursive call on a 0-length array and one on an array of length *i-1*. That's *n* levels.

$O(n^2)$

This is just selection sort!

Complexity of Quicksort

QUICKsort ?!

A blatant case of false advertising?

- Worst-case complexity is $O(n^2)$
  - if array is (largely) already sorted
- Best case complexity is $O(n \log n)$
  - if we are so lucky to pick the median each time as the pivot
- What happens on average?
  - if we add up the cost for *each possible input* and divide by the number of possible inputs

The details are beyond the scope of this class

O(n log n)

This is what we expect if the array contains values selected at random

- but we may be unlucky and get $O(n^2)$!

- This is called **average-case complexity**

Complexity of Quicksort

- Worst-case complexity is $O(n^2)$
  - if array is (largely) already sorted
- Best case complexity is $O(n \log n)$
  - if we are so lucky to pick the median each time as the pivot
- Average-case complexity is $O(n \log n)$
  - **if we are not too unlucky**

- In practice, quicksort is pretty fast,
  - it often outperforms mergesort
  - and it is in-place!

quicksort ?!

Maybe there is something to it …

Selecting the Pivot

- How is the pivot chosen in practice?

- Common ways:
  - Pick A[lo]
    - or the element at any fixed index

  - Choose an index i at **random** and pick A[i]

  - Choose 3 indices i1, i2 and i3,
    and pick the median of A[i1], A[i2] and A[i3]

Comparing Sorting Algorithms

- Three algorithms to solve the **same problem**
  - ➢ and there are many more!
  - ○ mergesort is asymptotically faster: *O(n log n)* vs. $O(n^2)$
  - ○ selection sort and quicksort are in-place but merge sort is not
  - ○ quicksort is ***on average*** as fast as mergesort

	Selection sort	**Mergesort**	**Quicksort**
Worst-case complexity	$O(n^2)$	*O(n log n)*	$O(n^2)$
In-place?	Yes	No	Yes
Average-case complexity	$O(n^2)$	*O(n log n)*	*O(n log n)*

- *Exercises:*
  - ○ *Check that selection sort and mergesort have the given average-case complexity*
    - ➢ *Hint: there is no luck involved*

Stable Sorting

Sorting in Practice

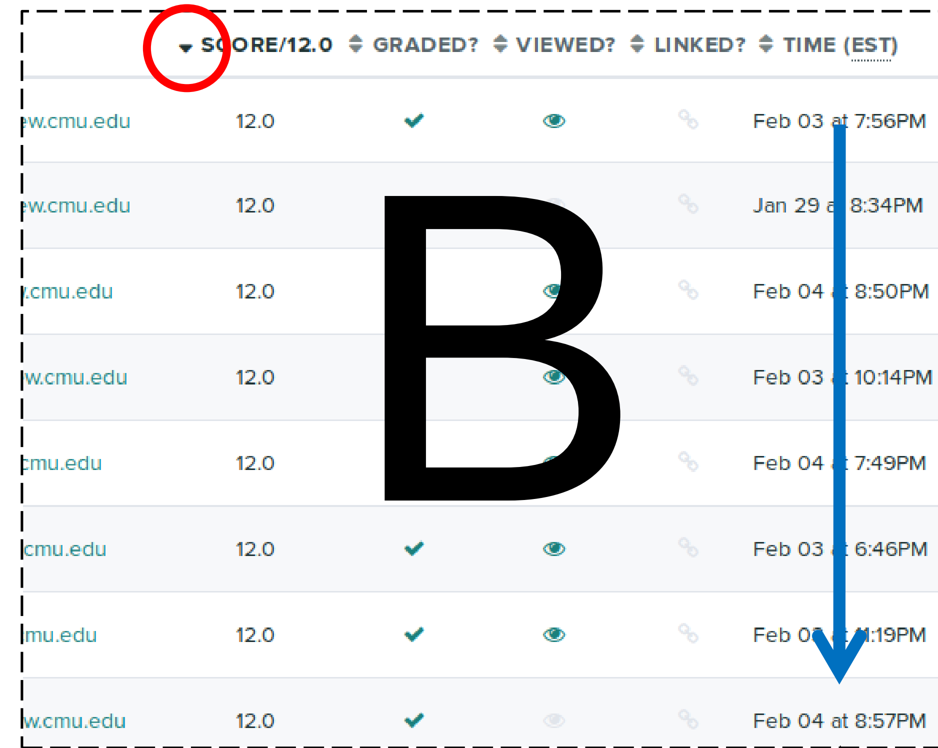
- We are not interested in sorting just numbers
  - also strings, characters, etc
- and **records**
  - e.g., student records in tabular form

sorting algorithm

	SCORE/12.0	GRADED?	VIEWED?	LINKED?	TIME (EST)
ndrew.cmu.edu	11.0	✔			Feb 04 at 12:16PM
andrew.cmu.edu	11.25	✔			Feb 04 at 12:43AM
drew.cmu.edu	7.75	✔			Feb 04 at 8:34PM
andrew.cmu.edu	10.25	✔			Feb 04 at 8:47PM
@andrew.cmu.edu	10.5	✔			Feb 04 at 5:55PM
andrew.cmu.edu	11.25	✔			Feb 04 at 12:29AM
andrew.cmu.edu	10.05	✔			Feb 04 at 4:06PM
@andrew.cmu.edu	10.5	✔			Feb 03 at 6:29PM
@andrew.cmu.edu	11.25	✔			Feb 04 at 7:24PM

Stability

- Say the table is already sorted by time and we sort it by score



	SCORE/12.0	GRADED?	VIEWED?	LINKED?	TIME (EST)
w.cmu.edu	12.0	✓			Feb 03 at 7:56PM
w.cmu.edu	12.0				Jan 29 at 8:34PM
.cmu.edu	12.0				Feb 04 at 8:50PM
w.cmu.edu	12.0				Feb 03 at 10:14PM
mu.edu	12.0				Feb 04 at 7:49PM
cmu.edu	12.0	✓			Feb 03 at 6:46PM
mu.edu	12.0	✓			Feb 0[illegible]:19PM
w.cmu.edu	12.0	✓			Feb 04 at 8:57PM

time ordering is **not preserved** for any given score

- Two possible outcomes:
  A. relative time order within each score is preserved
  B. relative time order within each score is lost

- A sorting algorithm that always does A is called **stable**
  - stable sorting is desirable for spreadsheets and other consumer-facing applications
  - it is irrelevant for some other applications

- New parameter to consider when choosing sorting algorithms

Stability

- In general,

  a sorting algorithm is **stable** if the relative order of duplicate elements doesn't change after sorting

  - the 1st occurrence of x in the input array is the 1st occurrence of x in the sorted array
  - the 2nd occurrence of x is till the 2nd occurrence
  - etc

Comparing Sorting Algorithms

- Three algorithms to solve the **same problem**
  - mergesort is asymptotically faster: $O(n \log n)$ vs. $O(n^2)$
  - selection sort and quicksort are in-place but merge sort is not
  - quicksort is on average as fast as mergesort
  - mergesort is stable

	Selection sort	**Mergesort**	**Quicksort**
Worst-case complexity	$O(n^2)$	$O(n \log n)$	$O(n^2)$
In-place?	Yes	No	Yes
Average-case complexity	$O(n^2)$	$O(n \log n)$	$O(n \log n)$
Stable?	No	Yes	No

- *Exercises:*
  - *check that mergesort is stable*
  - *check that selection sort and quicksort are not*