CSCI 241

Lecture 6

Quicksort

Stability, In-Place Sorts

Announcements

- A1 due in one week.
- Quiz 1 scores will be increased 1 pt due to vague wording in question 1(a).
- Happenings around the department:
  - Monday, 10/8 – CSCI Resume Workshop presented by Filip Jagodzinski! – 5 pm in CF 110
  - Tuesday, 10/9 – ACM Ice Cream Social – 5 pm in CF 316
  - Tuesday, 10/9 – First Whiteboard Coders Meeting – 5 pm in CF 420
  - Wednesday, 10/10 – The Game of Cybersecurity, presented by Shay Colson– 5 pm in CF 125
  - Wednesday and Thursday, 10/10 & 10/11 – Google is on Campus! Check their agenda here!

Goals:

- Thoroughly understand the mechanism of mergesort and quicksort.
- Be prepared to implement **merge** and **partition** helper methods.
- Know how to determine whether a sort is **in-place** and **stable**.

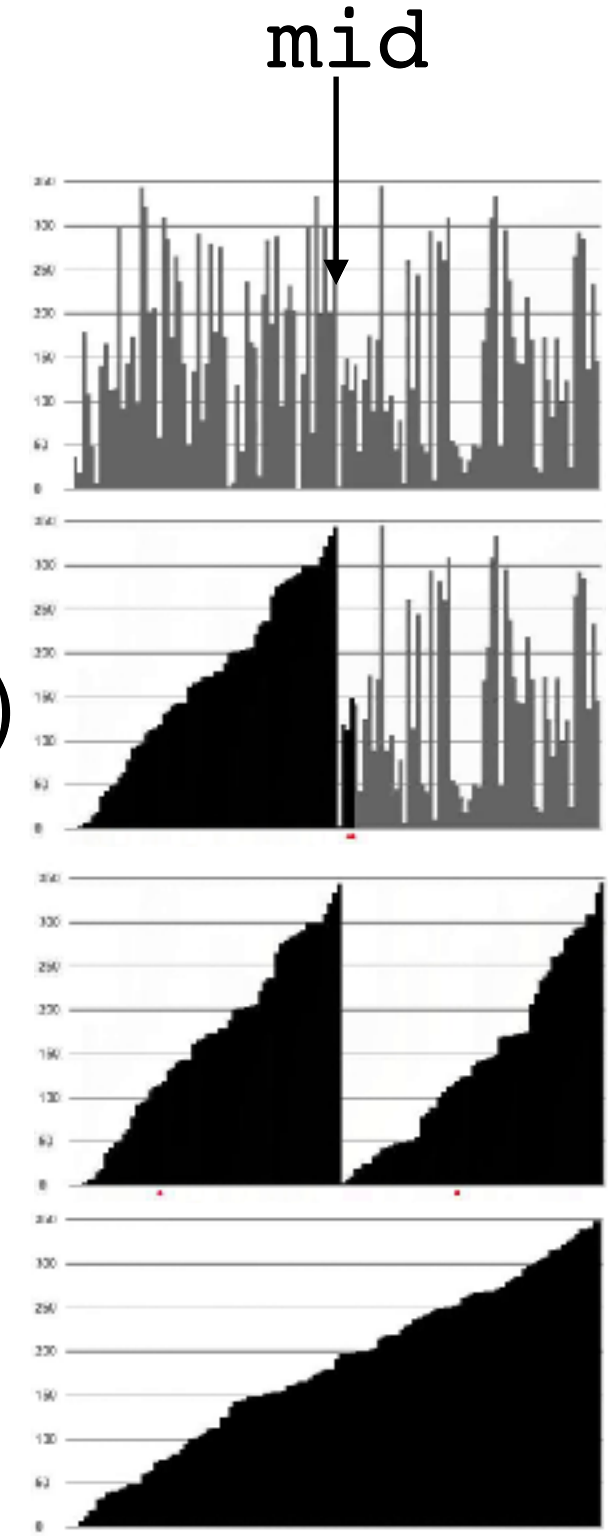
```
/** sort A[start..end] using mergesort */
mergeSort(A, start, end):
  if (end-start < 2):
    return
  mid = (end-start)/2

  mergeSort(A,start,mid)

  mergeSort(A,mid, end)

  merge(A, start, mid, end)
```

Divide

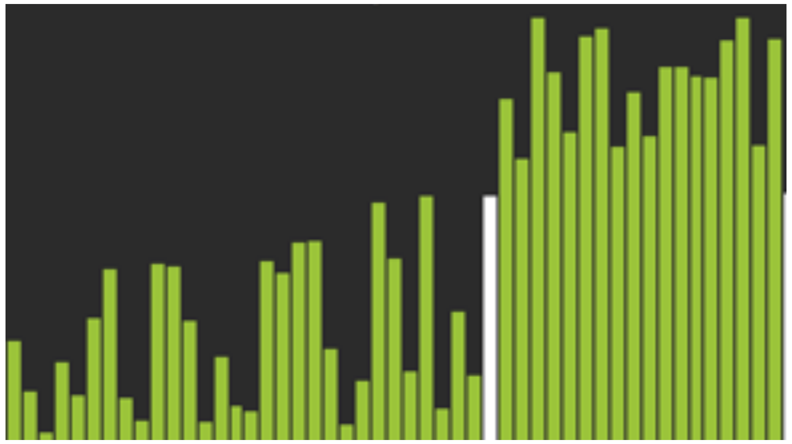
Conquer (left)

Conquer (right)

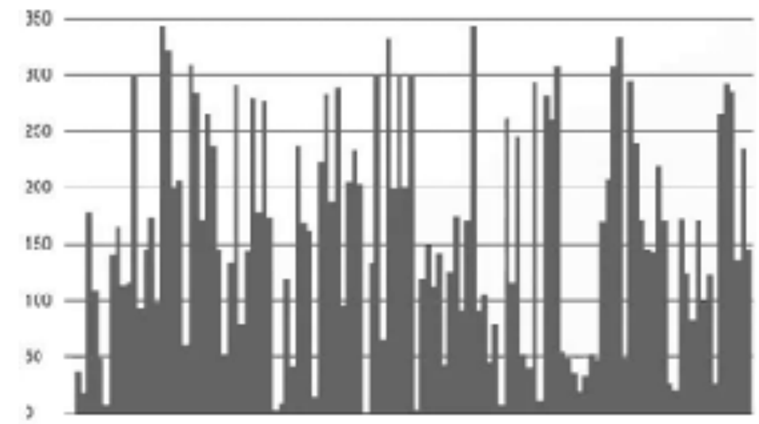
Combine

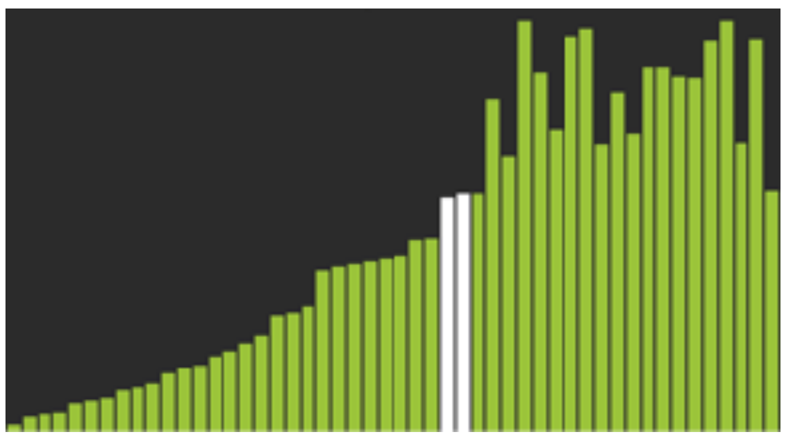


Quicksort | **Mergesort**

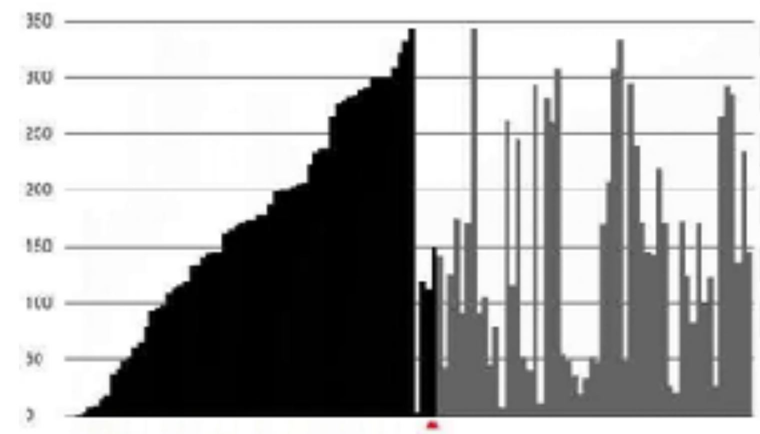


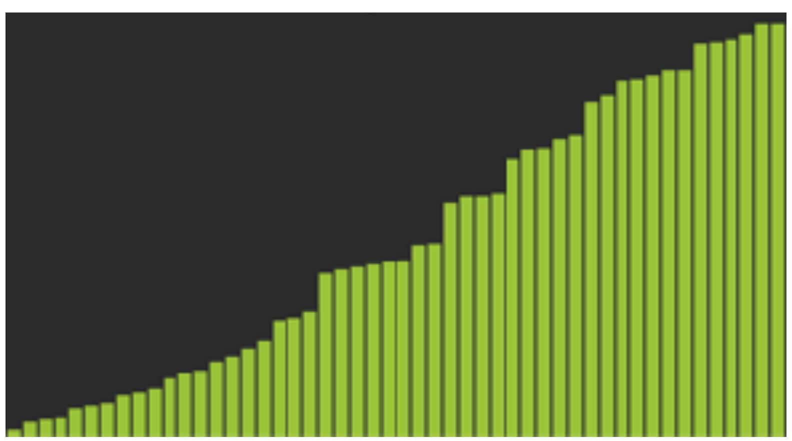
Divide



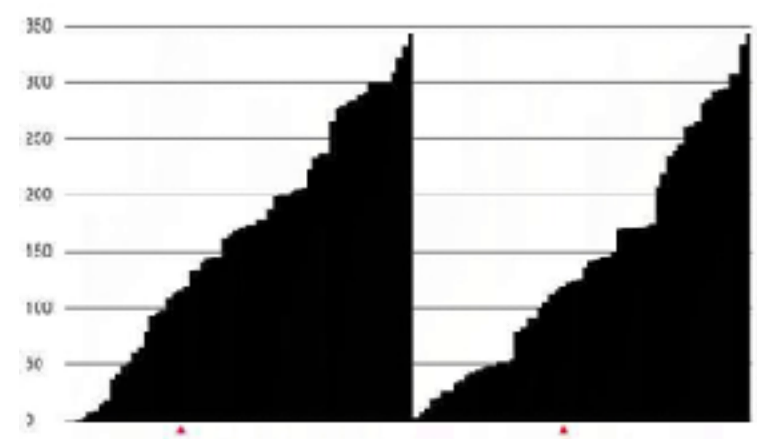


Conquer (left)





Conquer (right)



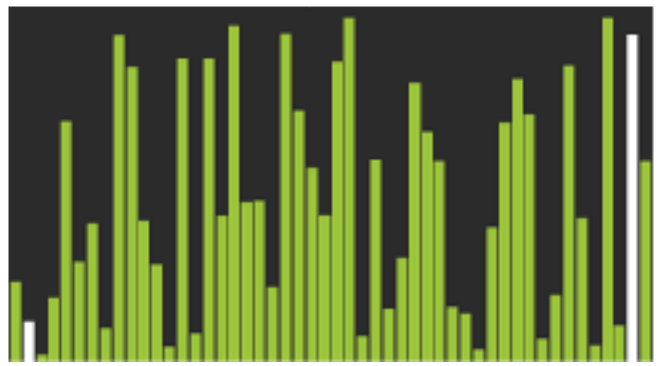
(done!)

Combine

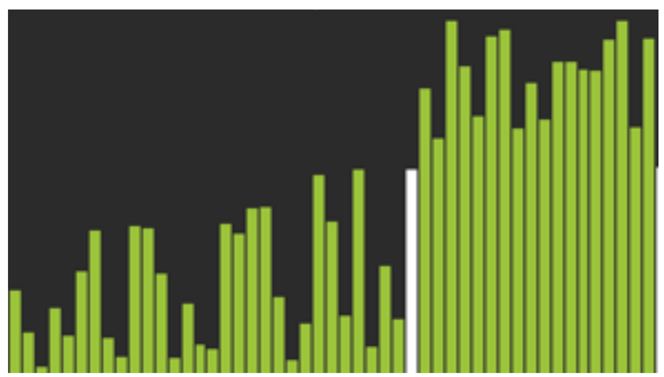
Quicksort

```
/** quicksort A[st..end]*/
quickSort(A, st, end):
  if (small):
    return
```

Unsorted:

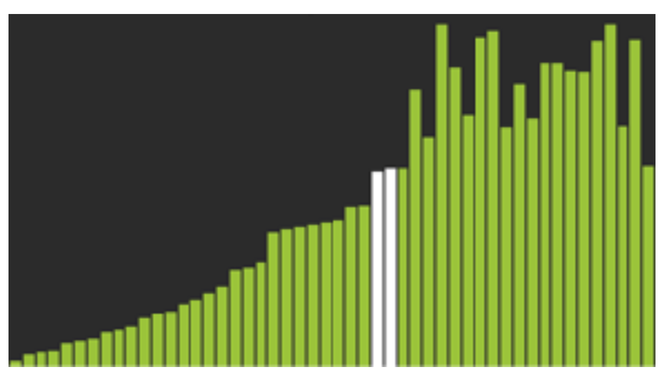


Smaller things left
bigger things right:



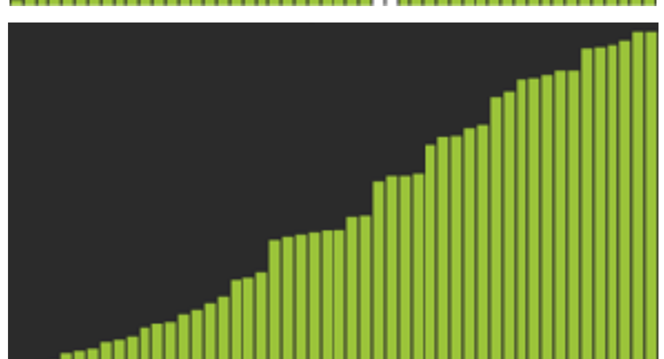
```
← mid = partition(A,st,end)
```

Sort left things:



```
← quickSort(A,st,mid)
```

Sort right things:



```
← quickSort(A,mid, end)
```

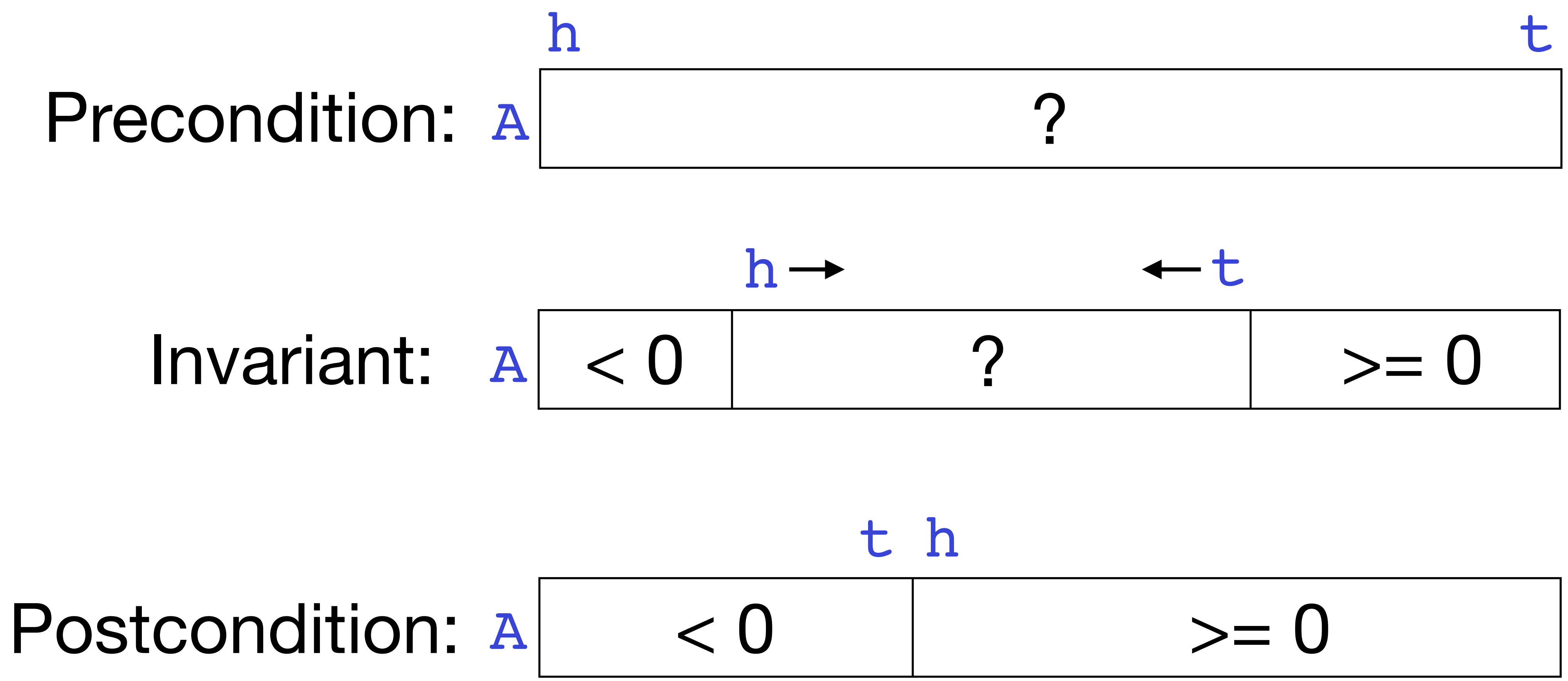
Quicksort

Key issues:

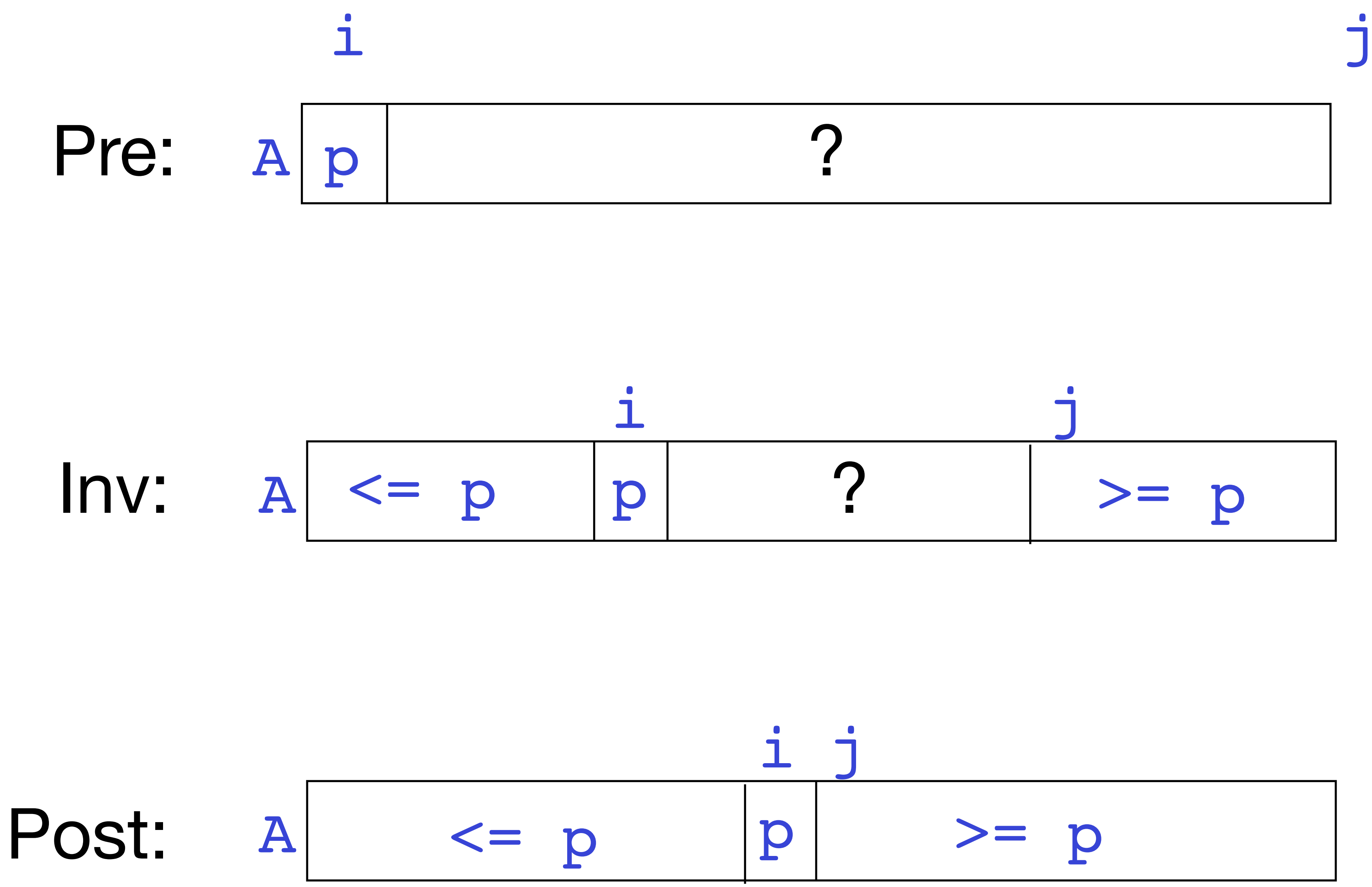
1. Implementing `partition`

```
/** quicksort A[st..end]*/
quickSort(A, st, end):
  if (small):
    return

  mid = partition(A,st,end)

  quickSort(A,st,mid)

  quickSort(A,mid, end)
```

```
/** rearrange A so all negative values are to
  * the left of all non-negative values */
public void separateSign(int[] A) {
```



```java
/** partition A around the pivot A[pivIndex].
  * return the pivot's new index.
  *  precondition: start <= pivIndex < end
  *  postcondition: A[start..i] <= A[i] <= A[i+1..end]
  *     where i is the return value */
    public int partition(int[] A, int start, int end, int pivIndex) {
```



Four concerns:

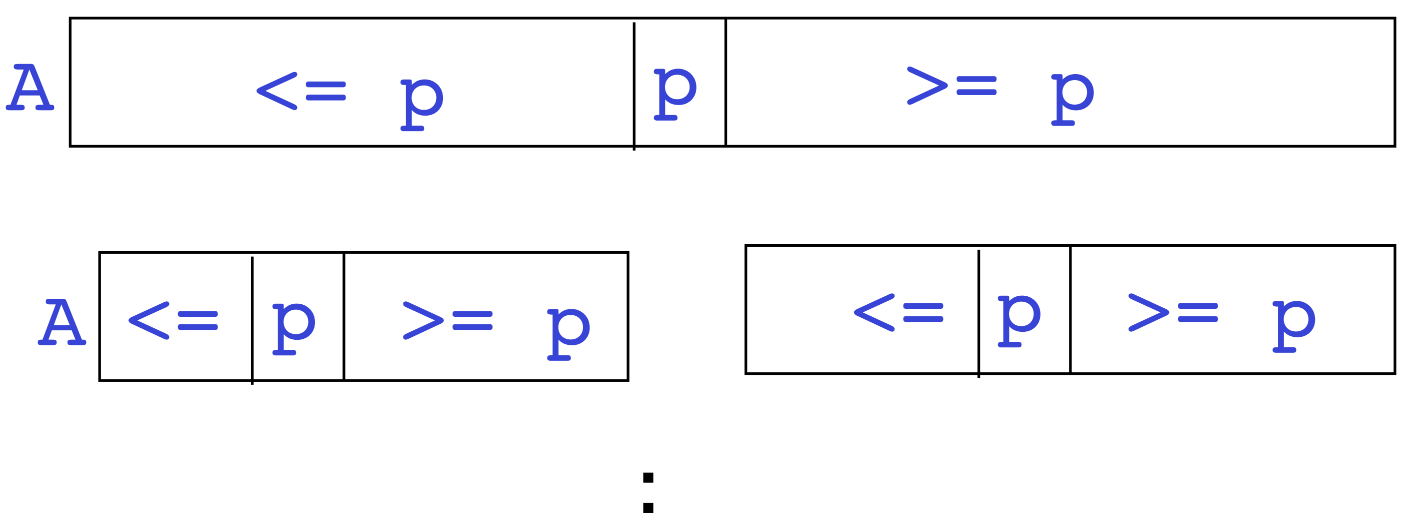
1. **Initialization**
2. **Termination**
3. **Progress**
4. **Maintenance**

Quicksort

Key issues:

1. Implementing `partition`

2. Runtime?




```
/** quicksort A[st..end]*/
quickSort(A, st, end):
  if (small):
    return

  mid = partition(A,st,end)

  quickSort(A,st,mid)

  quickSort(A,mid, end)
```

Quicksort

Key issues:

1. Implementing `partition`

2. Runtime?

```
/** quicksort A[st..end]*/
quickSort(A, st, end):
  if (small):
    return

  mid = partition(A,st,end)

  quickSort(A,st,mid)

  quickSort(A,mid, end)
```

Quicksort

Key issues:

1. Implementing `partition`

2. Runtime?

```
/** quicksort A[st..end]*/
quickSort(A, st, end):
  if (small):
    return
```

O(1)

```
  mid = partition(A,st,end)
```

O(hmm)

```
  quickSort(A,st,mid)
```

O(huh?)

```
  quickSort(A,mid, end)
```

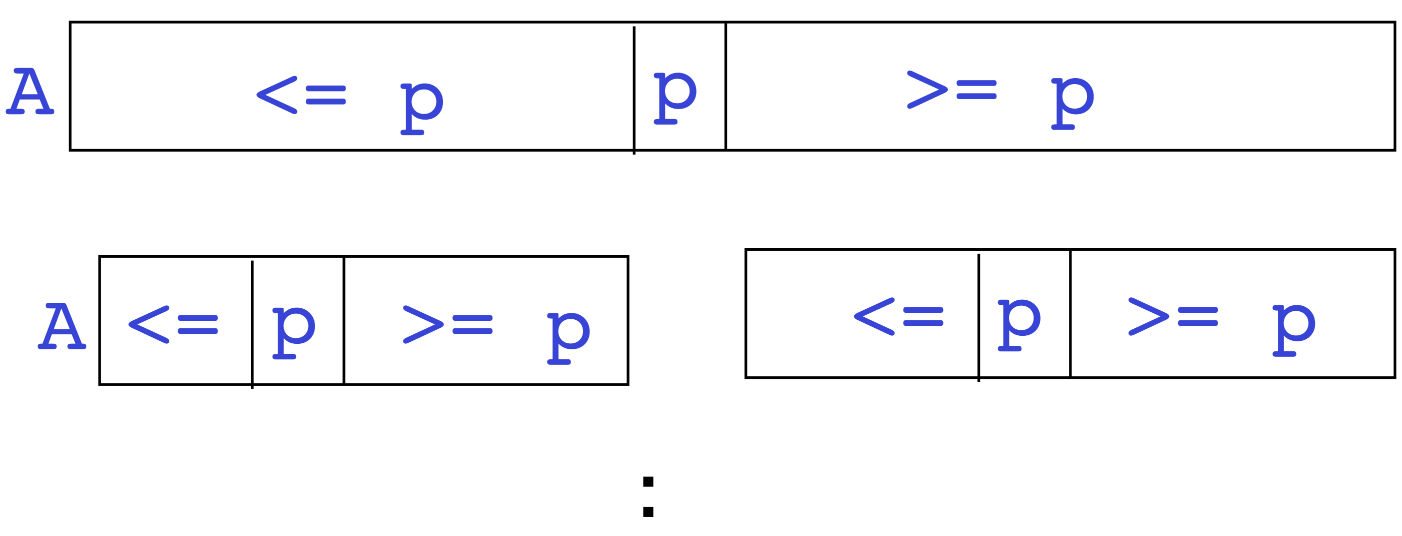
O(huh?)

Quicksort

Key issues:

1. Implementing `partition`

2. Runtime?




```
/** quicksort A[st..end]*/
quickSort(A, st, end):
  if (small):
    return
```

O(1)

```
  mid = partition(A,st,end)
```

O(hmm)

```
  quickSort(A,st,mid)
```

O(huh?)

```
  quickSort(A,mid, end)
```

O(huh?)

Quicksort

Key issues:

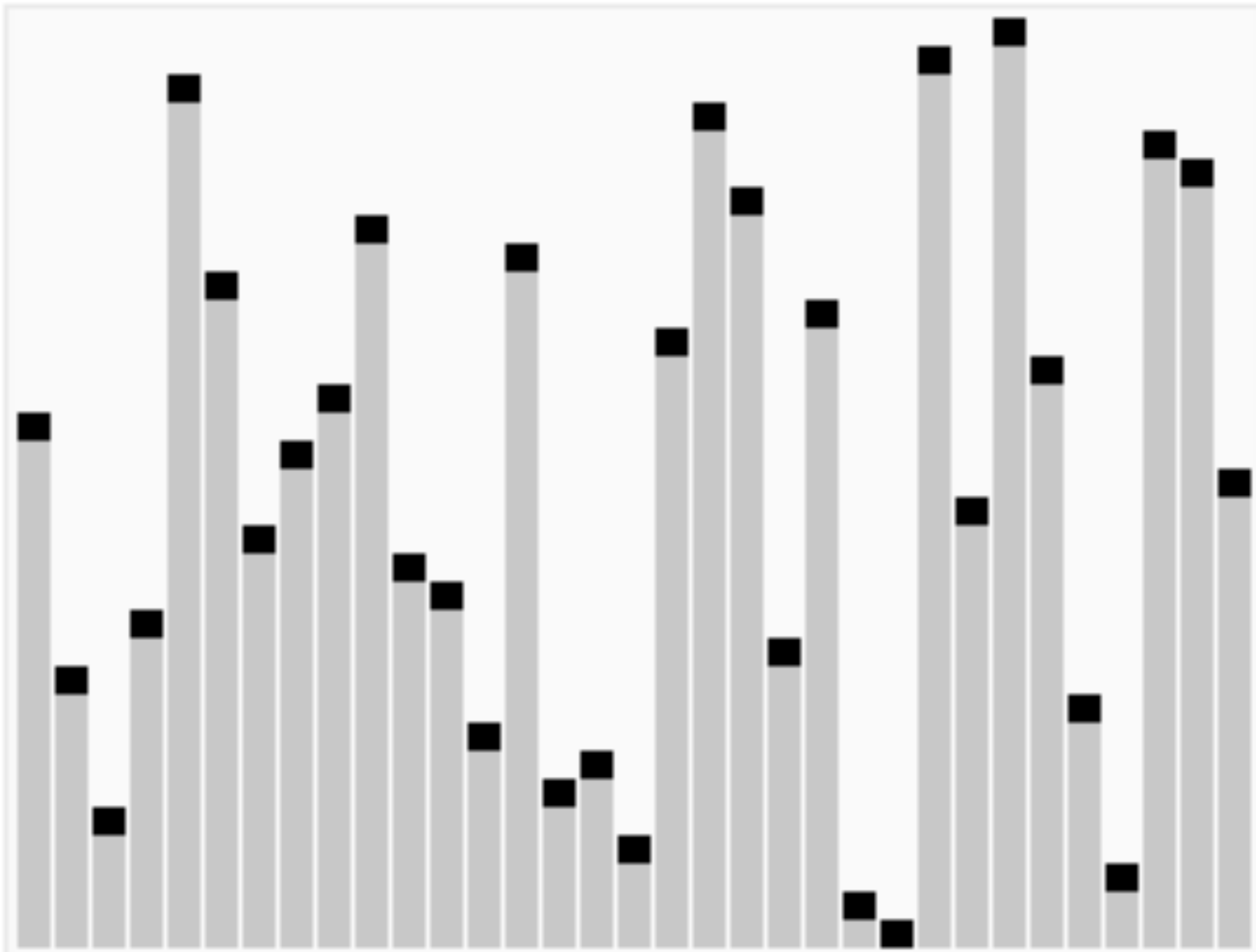
1. Implementing `partition`

2. Runtime?

3.Picking the pivot

- First, middle, or last
- Median of first, middle, or last

```
/** quicksort A[st..end]*/
quickSort(A, st, end):
  if (small):
    return

  mid = partition(A,st,end)

  quickSort(A,st,mid)

  quickSort(A,mid, end)
```



https://upload.wikimedia.org/wikipedia/commons/6/6a/Sorting_quicksort_anim.gif

In-Place

- Time complexity: how many operations?
- Space complexity: how much (extra) memory?
  - Usually don’t count the size of the input, because we have no choice but to store it.

In-Place

- Time complexity: how many operations?
- Space complexity: how much (extra) memory?
  - Usually don't count the size of the input, because we have no choice but to store it.

ABCD:
How much extra space does insertion sort use?

A. O(1)
B. O(log n)
C. O(n)
D. $O(n^2)$

```
insertionSort(A):
  i = 0;
  while i < A.length:
    j = i;
    while j > 0 and A[j] > A[j-1]:
      swap(A[j], A[j-1])
      j--
    i++
```

In-Place

A sort is considered **in-place** if it requires O(1) storage space in addition to the input.

ABCD:
How much extra space does insertion sort use?

A. O(1)
B. O(log n)
C. O(n)
D. O(n^2)

```
insertionSort(A):
  i = 0;
  while i < A.length:
    j = i;
    while j > 0 and A[j] > A[j-1]:
      swap(A[j], A[j-1])
      j--
    i++
```

Stability

Objects can be sorted on **keys** - **different** objects may have the same value.

- e.g., sorting on first name only.

A **stable** sort maintains the order of distinct elements with the same key.

Stability

A **stable** sort maintains the order of elements with the same value.

[6^* 2^* 6^+ 2^+ 3 4]

Stably sorted: **[2^* 2^+ 3 4 6^* 6^+]**

Unstably sorted: **[2^+ 2^* 3 4 6^* 6^+]**

Stability

A **stable** sort maintains the order of elements with the same value.

In groups: determine stability of insertionSort and selectionSort

$$[\ 6^{*}\quad 2^{*}\quad 6^{+}\quad 2^{+}\quad 3\quad 4\ ]$$

Stability

A **stable** sort maintains the order of elements with the same value.

Homework: Sort this list using insertion and selection sort. For each sort, write the state of the list after each iteration of the **outer** loop. For each sort, write whether it is stable or not.

$$[\,6^{*}\ \ 2^{*}\ \ 6^{+}\ \ 2^{+}\ \ 3\ \ 4\,]$$

```
insertionSort(A):
 i = 0;
 while i < A.length:
   // push A[i] to
   //  its sorted position
   //  in A[0..i]
   // increment i
```

```
selectionSort(A):
  i = 0;
  while i < A.length:
    // find min of A[i..A.length]
    // swap it with A[i]
    // increment i
```