"Organizing is what you do before you do something, so that when you do it, it is not all mixed up."

~ A. A. Milne

# SORTING

# Prelim 1: Tuesday, 12 March

Visit exams page of course website.

It tells you your assigned time to take it (5:30 or 7:30) and what to do if you have a conflict.

Anyone with any kind of a conflict *must* complete assignment

 P1 Conflict

on the CMS by midnight of Wednesday, 6 March.

It is extremely important that this be done correctly and on time. We have to schedule room and proctors and know how many of each prelim (5:30 or 7:30) to print.

# Recitation next week

Review for prelim!

# Why Sorting?

- Sorting is useful
  - Database indexing
  - Operations research
  - Compression
- There are lots of ways to sort
  - There isn't one right answer
  - You need to be able to figure out the options and decide which one is right for your application.
  - Today, we'll learn several different algorithms (and how to develop them)

# We look at four sorting algorithms

- Insertion sort

- Selection sort

- Quick sort

- Merge sort

# InsertionSort

pre: b
0          b.length
| ? |

post: b
0          b.length
| sorted |

inv: b
0          i          b.length
| sorted | ? |

or:      b[0..i-1] is sorted

inv: b
0          i          b.length
| processed | ? |

or:      b[0..i-1] is processed

A loop that processes elements of an array in increasing order has this invariant --- just replace "sorted" by "processed".

# Each iteration, i= i+1; How to keep inv true?

inv:

```
        0                    i              b.length
   b  ┌──────────────────┬─────────────────────┐
      │     sorted       │          ?          │
      └──────────────────┴─────────────────────┘
```

e.g.

```
        0                    i              b.length
   b  ┌──────────────────┬─────────────────────┐
      │  2  5  5  5  7   │  3     ?            │
      └──────────────────┴─────────────────────┘
```

```
        0                        i          b.length
   b  ┌──────────────────────┬─────────────────┐
      │  2  3  5  5  5  7    │        ?        │
      └──────────────────────┴─────────────────┘
```

# What to do in each iteration?

inv:   b

| 0 | | i | | b.length |
|---|---|---|---|---|
| sorted | | ? | | |

e.g.   b

| 0 | | | | | i | | b.length |
|---|---|---|---|---|---|---|---|
| 2 | 5 | 5 | 5 | 7 | 3 | ? | |

Loop body (inv true before and after)

| 2 | 5 | 5 | 5 | 3 | 7 | ? |
|---|---|---|---|---|---|---|
| 2 | 5 | 5 | 3 | 5 | 7 | ? |
| 2 | 5 | 3 | 5 | 5 | 7 | ? |
| 2 | 3 | 5 | 5 | 5 | 7 | ? |

Push b[i] to its sorted position in b[0..i], then increase i

This will take time proportional to the number of swaps needed

b
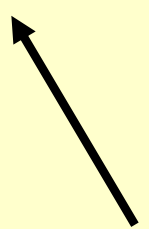
| 2 | 3 | 5 | 5 | 5 | 7 | ? |
|---|---|---|---|---|---|---|

# Insertion Sort

```
// sort b[], an array of int
// inv: b[0..i-1] is sorted
for (int i= 0; i < b.length; i= i+1) {
    // Push b[i] down to its sorted
    // position in b[0..i]



}
```

Present algorithm like this

Note English statement in body.

**Abstraction**. Says **what** to do, not **how.**

This is the best way to present it. We expect you to present it this way when asked.

Later, can show how to implement that with an inner loop

# Insertion Sort

```
// sort b[], an array of int
// inv: b[0..i-1] is sorted
for (int i= 0; i < b.length; i= i+1) {
    // Push b[i] down to its sorted
    // position in b[0..i]
    int k= i;
    while (k > 0  &&  b[k] < b[k-1]) {
        Swap b[k] and b[k-1];
        k= k–1;
    }
}
```

invariant P:  b[0..i] is sorted
**except** that b[k] may be < b[k-1]

```
       k        i
| 2  5  3  5  5 | 7   ? |
```
example

start?

stop?

progress?

maintain
invariant?

# Insertion Sort

```
// sort b[], an array of int
// inv: b[0..i-1] is sorted
for (int i= 0; i < b.length; i= i+1) {
    // Push b[i] down to its sorted
    // position in b[0..i]}
```

Pushing b[i] down can take i swaps.
Worst case takes

$$1 + 2 + 3 + \ldots n-1 = (n-1)*n/2$$

swaps.

Let n = b.length

- Worst-case: $O(n^2)$
  (reverse-sorted input)

- Best-case: $O(n)$
  (sorted input)

- Expected case: $O(n^2)$

# Insertion Sort is stable

```
// sort b[], an array of int
// inv: b[0..i-1] is sorted
for (int i= 0; i < b.length; i= i+1) {
    // Push b[i] down to its sorted
    // position in b[0..i]}
```

Let n = b.length

A sorting algorithm is stable if two equal values stay in the same relative position.

initial: (3 7, 2 8, 7, 6)

stably sorted (2, 3, 6, 7, 7, 8)

unstably sorted (2, 3, 6, 7, 7, 8)

**Insertion sort is stable**

# Performance

| Algorithm | Ave time. | Worst-case time | Space | Stable? |
|-----------|-----------|-----------------|-------|---------|
| Insertion Sort | $O(n^2)$. | $O(n^2)$ | $O(1)$ | Yes |
| | | | | |
| | | | | |

# SelectionSort

pre: b  0 [ ? ] b.length

post: b  0 [ sorted ] b.length

inv: b  0 [ sorted, <= b[i..] | >= b[0..i-1] ] i  b.length

Additional term in invariant

Keep invariant true while making progress?

e.g.: b  0 [ 1  2  3  4  5  6 | 9  9  9  7  8  6  9 ] i  b.length

Increasing i by 1 keeps inv true only if b[i] is min of b[i..]

# SelectionSort

```
// sort b[].
// inv: b[0..i-1] sorted  AND
//       b[0..i-1] <= b[i..]
for (int i= 0; i < b.length; i= i+1) {
  int m= index of min of b[i..];
  Swap b[i] and b[m];
}
```

Another common way for people to sort cards

Runtime
with n = b.length
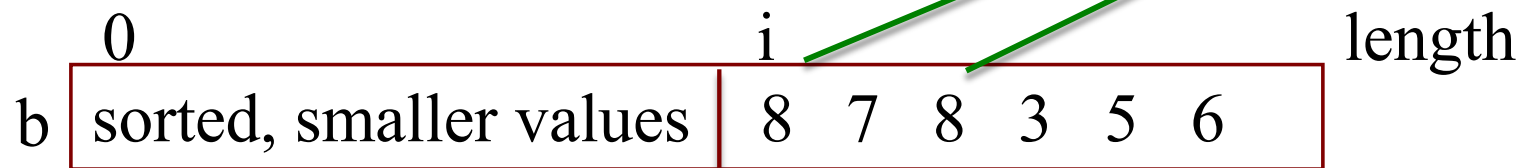- Worst-case $O(n^2)$
- Best-case $O(n^2)$
- Expected-case $O(n^2)$

```
      0                    i              length
b |  sorted, smaller values  |  larger values  |
```
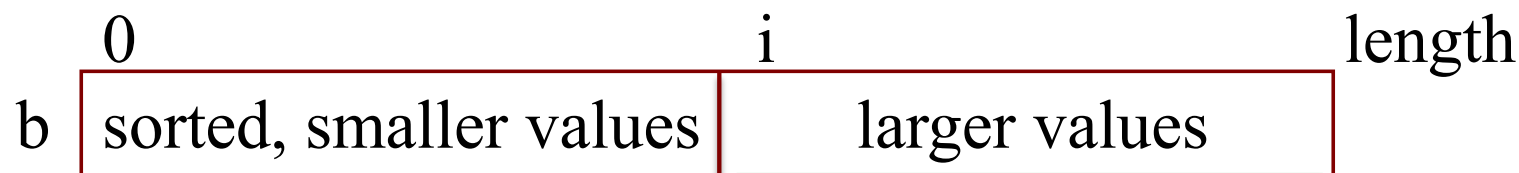
Each iteration, swap min value of this section into b[i]

# SelectionSort —not stable

```
// sort b[].
// inv: b[0..i-1] sorted  AND
//        b[0..i-1]  <=  b[i..]
for (int i= 0; i < b.length; i= i+1) {
    int m= index of min of b[i..];
    Swap b[i] and b[m];
}
```

Here, swapping b[i] with the minimum of b[i..], 3, changes the relative position of the two 8's.

```
      0                        i              length
 b |  sorted, smaller values |  8  7  8  3  5  6 |
```

# Performance

| Algorithm | Ave time. | Worst-case time | Space | Stable? |
|---|---|---|---|---|
| Insertion sort | $O(n^2)$ | $O(n^2)$ | $O(1)$ | Yes |
| Selection sort | $O(n^2)$ | $O(n^2)$ | $O(1)$ | No |
| | | | | |
| | | | | |

# QuickSort

Quicksort developed by Sir Tony Hoare (he was knighted by the Queen of England for his contributions to education and CS).

84 years old.

Developed Quicksort in 1958. But he could not explain it to his colleague, so he gave up on it.

Later, he saw a draft of the new language Algol 58 (which became Algol 60). It had recursive procedures. First time in a procedural programming language. "Ah!," he said. "I know how to write it better now." 15 minutes later, his colleague also understood it.
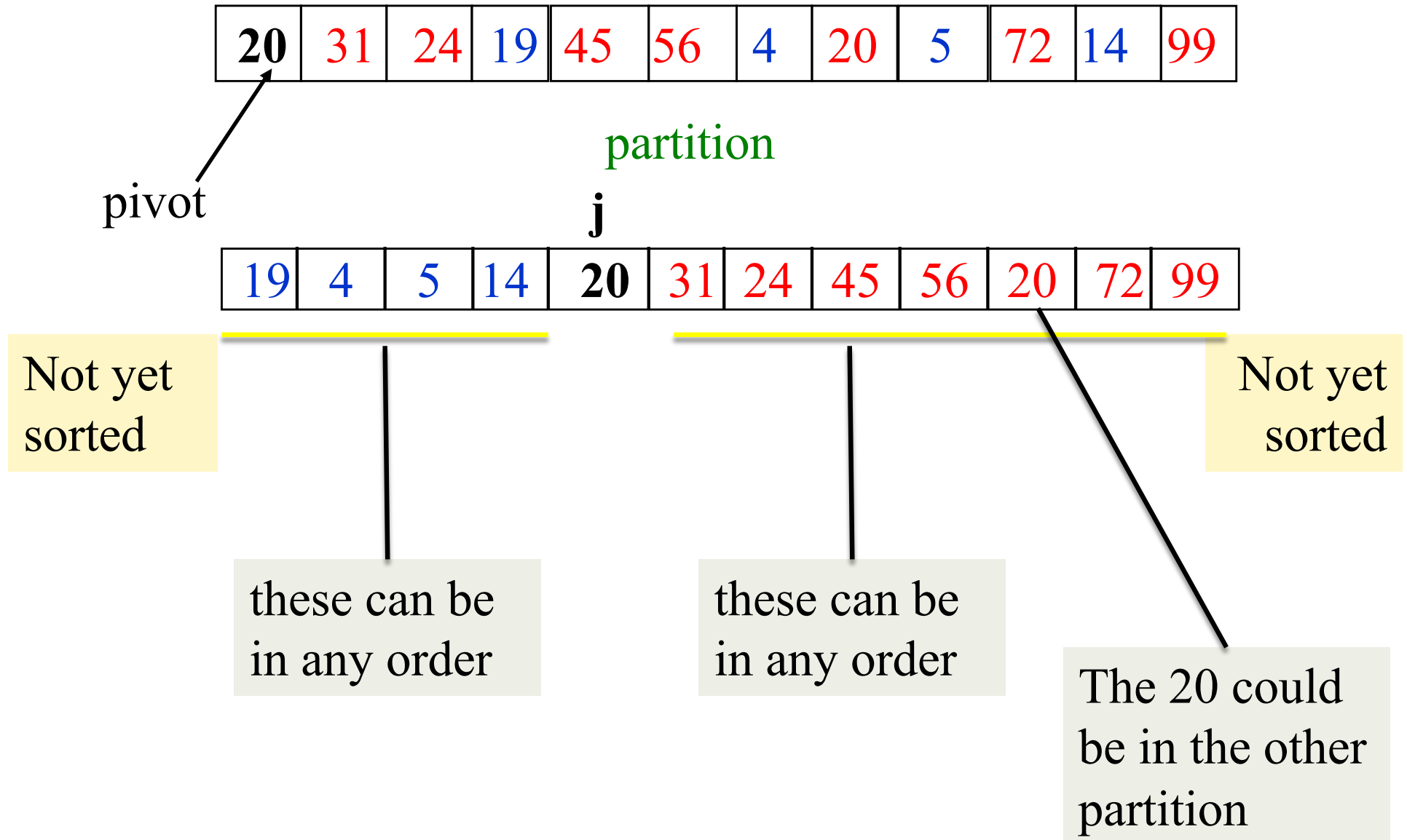
# Dijkstras, Hoares, Grieses, 1980s

# Partition algorithm of quicksort

h   h+1                                    k

pre:   | x |              ?              |

x is called
the pivot

Swap array values around until b[h..k] looks like this:

h                          j              k

post:   |      <= x      | x |   >= x   |

# Partition algorithm of quicksort

| 20 | 31 | 24 | 19 | 45 | 56 | 4 | 20 | 5 | 72 | 14 | 99 |
|----|----|----|----|----|----|---|----|---|----|----|----|

pivot

partition

**j**

| 19 | 4 | 5 | 14 | 20 | 31 | 24 | 45 | 56 | 20 | 72 | 99 |
|----|---|---|----|----|----|----|----|----|----|----|----|

Not yet sorted

Not yet sorted

these can be in any order

these can be in any order

The 20 could be in the other partition

# Partition algorithm

pre:

h   h+1                                k

b | x |                    ?                    |

post:

h                        j                  k

b |         <= x         | x |      >= x      |

Combine pre and post to get an invariant

h                  j            t        k

b |      <= x      | x |   ?   |   >= x   |
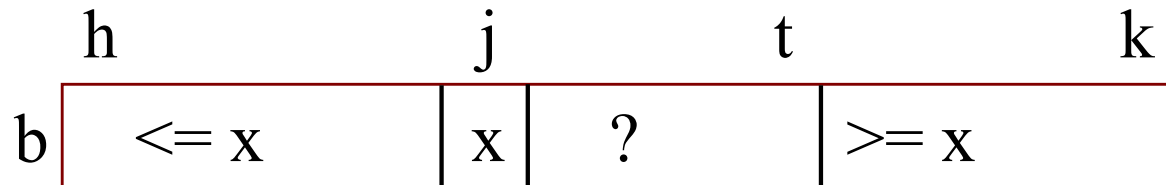
invariant needs at least 4 sections

# Partition algorithm

| h | | j | t | | k |
|---|---|---|---|---|---|
| b | <= x | x | ? | >= x | |

Initially, with j = h and t = k, this diagram looks like the start diagram
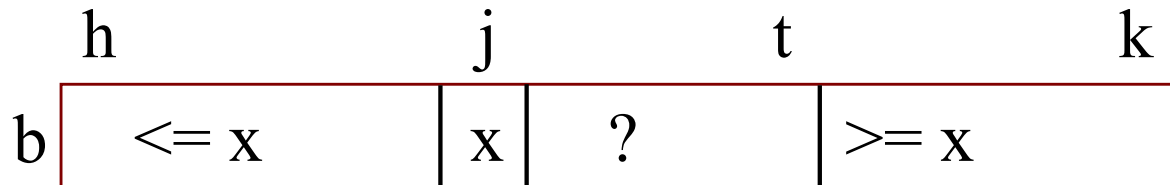
```
j= h; t= k;
while (j < t) {
    if (b[j+1] <= b[j]) {
        Swap b[j+1] and b[j];  j= j+1;
    } else {
        Swap b[j+1] and b[t];  t= t-1;
    }
}
```

Takes linear time: $O(k+1-h)$

Terminate when j = t, so the "?" segment is empty, so diagram looks like result diagram

# QuickSort procedure

```
/** Sort b[h..k]. */
public static void QS(int[] b, int h, int k) {
    if (b[h..k] has < 2 elements) return;        Base case

    int j= partition(b, h, k);
        // We know b[h..j–1] <= b[j] <= b[j+1..k]

        // Sort b[h..j-1] and b[j+1..k]

        QS(b, h, j-1);
        QS(b, j+1, k);
    }
```

Function does the partition algorithm and returns position j of pivot

| h | | j | | k |
|---|---|---|---|---|
| <= x | | x | >= x | |
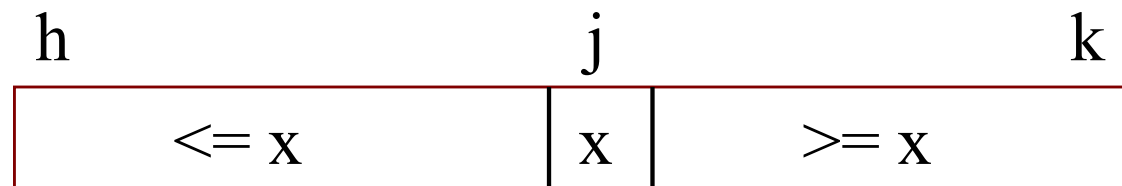
# Worst case quicksort: pivot always smallest value
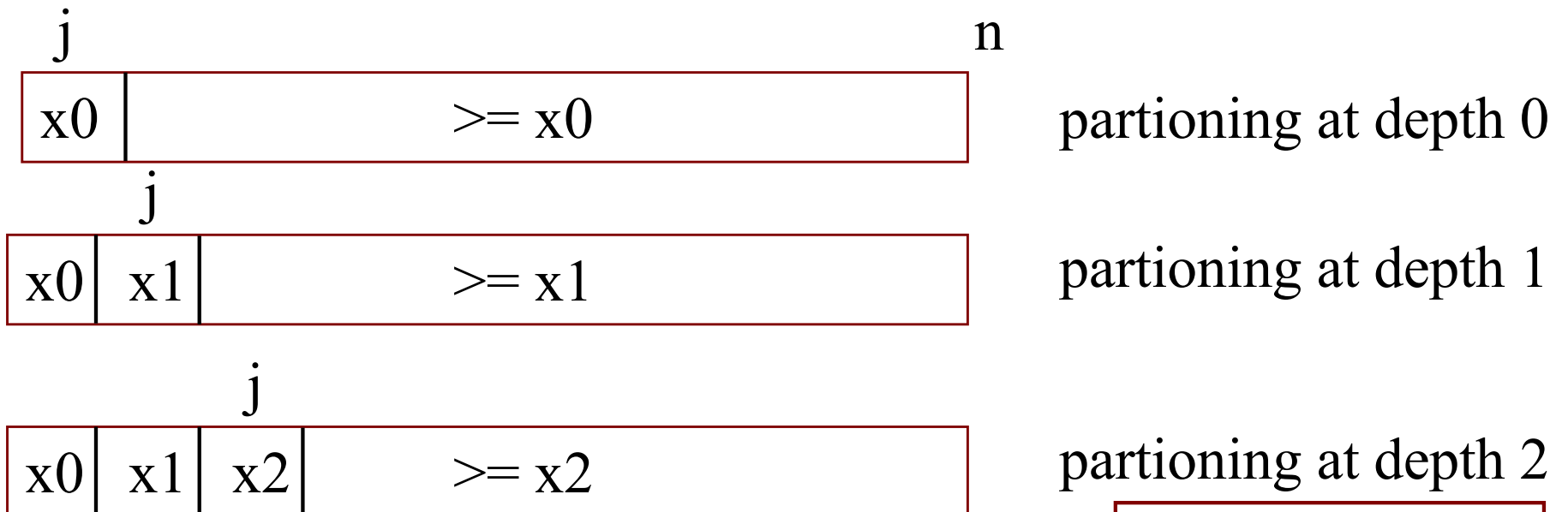
```
j                                          n
┌─────┬──────────────────────────────────┐
│ x0  │            >= x0                   │   partioning at depth 0
└─────┴──────────────────────────────────┘
    j
┌─────┬─────┬────────────────────────────┐
│ x0  │ x1  │          >= x1              │   partioning at depth 1
└─────┴─────┴────────────────────────────┘
          j
┌─────┬─────┬─────┬──────────────────────┐
│ x0  │ x1  │ x2  │        >= x2          │   partioning at depth 2
└─────┴─────┴─────┴──────────────────────┘
```

/** Sort b[h..k]. */
**public static void** QS(**int**[] b, **int** h, **int** k) {
   **if** (b[h..k] has < 2 elements) **return**;
   **int** j= partition(b, h, k);
   QS(b, h, j-1);   QS(b, j+1, k);

Depth of recursion: $O(n)$

Processing at depth i: $O(n-i)$

$O(n*n)$

# Best case quicksort: pivot always middle value

| 0 | | j | | n |
|---|---|---|---|---|
| $\leq x0$ | | x0 | $\geq x0$ | |
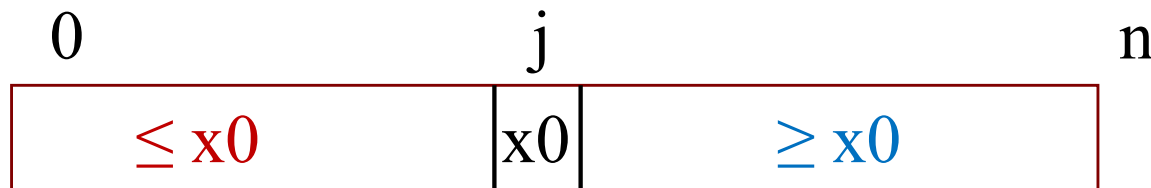
depth 0. 1 segment of size n to partition.

| $\leq x1$ | x1 | $\geq x1$ | x0 | $\leq x2$ | x2 | $\geq x2$ |
|---|---|---|---|---|---|---|
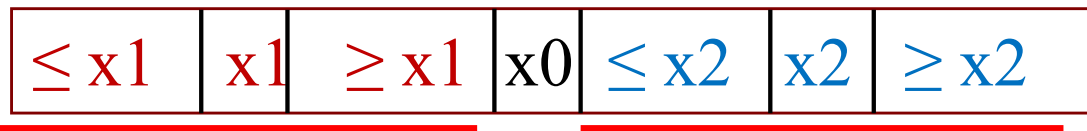
Depth 1. 2 segments of size $\leq n/2$ to partition.

Depth 2. 4 segments of size $\leq n/4$ to partition.

Max depth: O(log n).   Time to partition on each level: O(n)
Total time: O(n log n).

Average time for Quicksort: n log n. Difficult calculation

# QuickSort complexity to sort array of length n

Time complexity
Worst-case: O(n*n)
Average-case: O(n log n)

```
/** Sort b[h..k]. */
public static void QS(int[] b, int h, int k) {
    if (b[h..k] has < 2 elements) return;
    int j=  partition(b, h, k);
    // We know b[h..j–1] <= b[j] <= b[j+1..k]
    // Sort b[h..j-1] and b[j+1..k]
    QS(b, h, j-1);
    QS(b, j+1, k);
}
```

Worst-case space: ?
What's depth of recursion?

Worst-case space: O(n)!
   --depth of recursion can be n
Can rewrite it to have space O(log n)
Show this at end of lecture if we have time

# Partition. Key issue. How to choose pivot

```
            h                    k
pre:    b  x |        ?        |

            h           j        k
post:   b  |    <= x   | x | >= x |
```
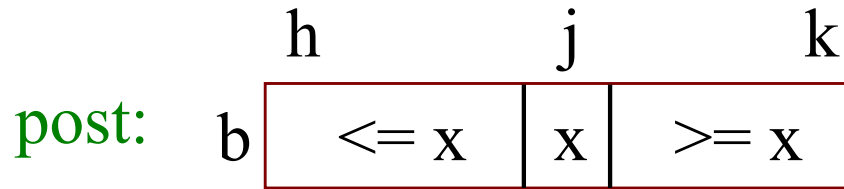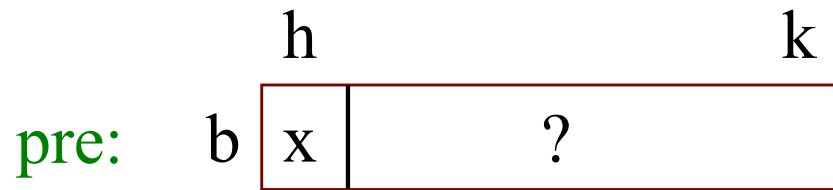
Choosing pivot

Ideal pivot: the median, since it splits array in half

But computing the median is O(n), quite complicated

Popular heuristics: Use
  ◆ first array value (not so good)
  ◆ middle array value (not so good)
  ◆ Choose a random element (not so good)
  ◆ median of first, middle, last, values (often used)!

# Performance

| Algorithm | Ave time. | Worst-case time | Space | Stable? |
|---|---|---|---|---|
| Insertion sort | $O(n^2)$ | $O(n^2)$ | $O(1)$ | Yes |
| Selection sort | $O(n^2)$ | $O(n^2)$ | $O(1)$ | No |
| Quick sort | $O(n \log n)$ | $O(n^2)$ | O(log n)* | No |
| Merge sort | | | | |

\* The first algorithm we developed takes space O(n) in the worst case, but it can be reduced to O(log n)
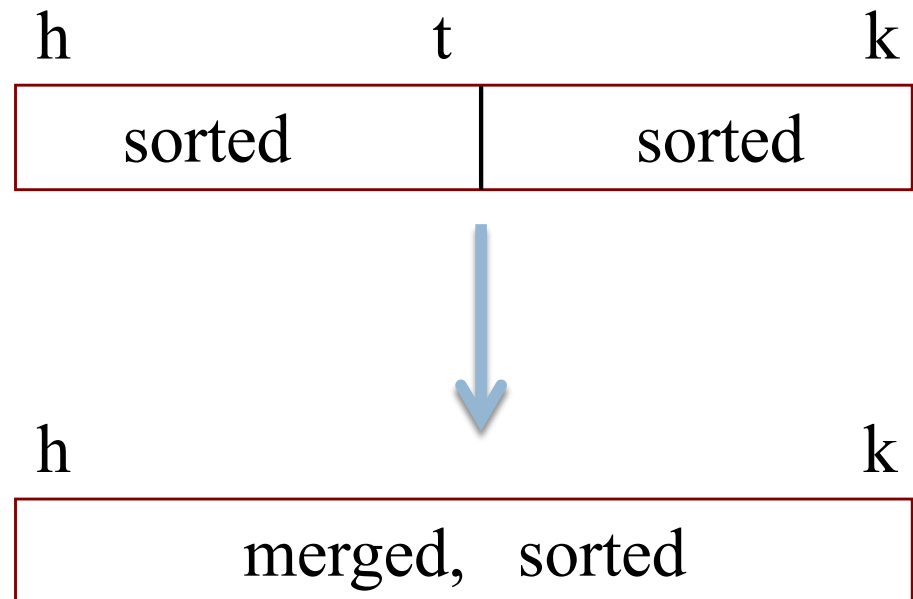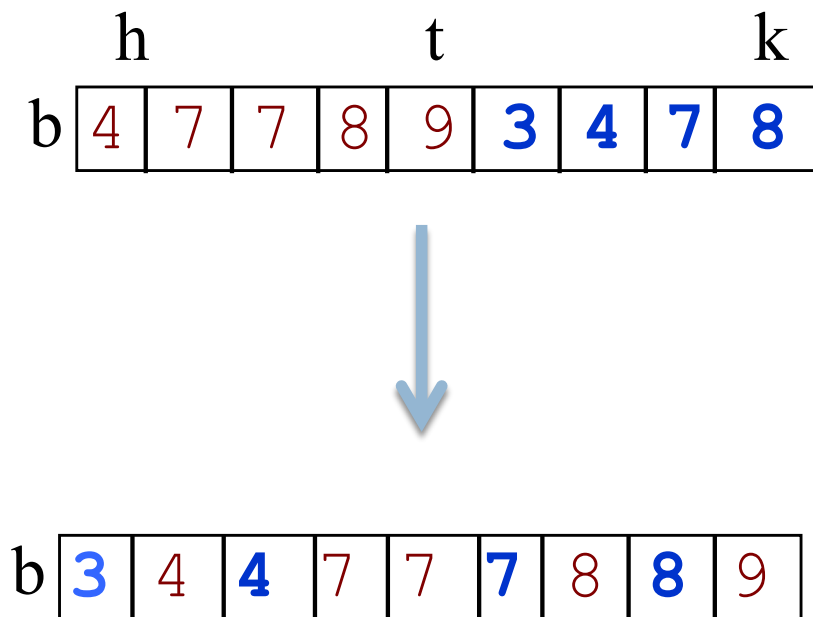
# Merge two adjacent sorted segments

/* Sort b[h..k]. Precondition: b[h..t] and b[t+1..k] are sorted.  */

public static merge(int[] b, int h, int t, int k) {

    Copy b[h..t] into a new array c;

    Merge c and b[t+1..k] into b[h..k];

}

# Merge two adjacent sorted segments

/* Sort b[h..k]. Precondition: b[h..t] and b[t+1..k] are sorted.  */

public static merge(int[] b, int h, int t, int k) {

}

h          t          k

b | 4 | 7 | 7 | 8 | 9 | **3** | **4** | **7** | **8** |

h                    t                    k

| sorted | sorted |

b | **3** | 4 | **4** | 7 | 7 | **7** | 8 | **8** | 9 |

h                                        k

| merged,  sorted |

# Merge two adjacent sorted segments

```
     0           t
c  | 4 | 7 | 7 | 8 | 9 |
```

```
     h           t           k
b  | ? | ? | ? | ? | ? | 3 | 4 | 7 | 8 |
```

Place all values in c[0..] and
b[t+1..k] into
b[h..k] in sorted order

```
     h               t               k
b  | 3 | 4 | 4 | 7 | 7 | 7 | 8 | 8 | 9 |
```

# Merge two adjacent sorted segments

```
    0
c [ 4 | 7 | 7 | 8 | 9 ]
```

```
    h           t           k
b [ ? | ? | ? | ? | ? | 3 | 4 | 7 | 8 ]
```

Place all values in c[0..] and b[t+1..k] into b[h..k] in sorted order
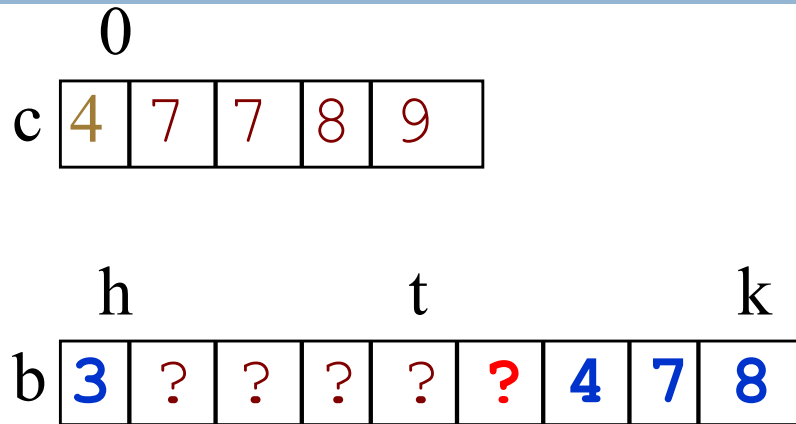
# Merge two adjacent sorted segments

0

c | 4 | 7 | 7 | 8 | 9 |

Step 1. Move **3**
from b[t+1] to b[h]

h       t       k

b | **3** | ? | ? | ? | ? | **?** | **4** | **7** | **8** |

# Merge two adjacent sorted segments

0

c | ? | 7 | 7 | 8 | 9 |
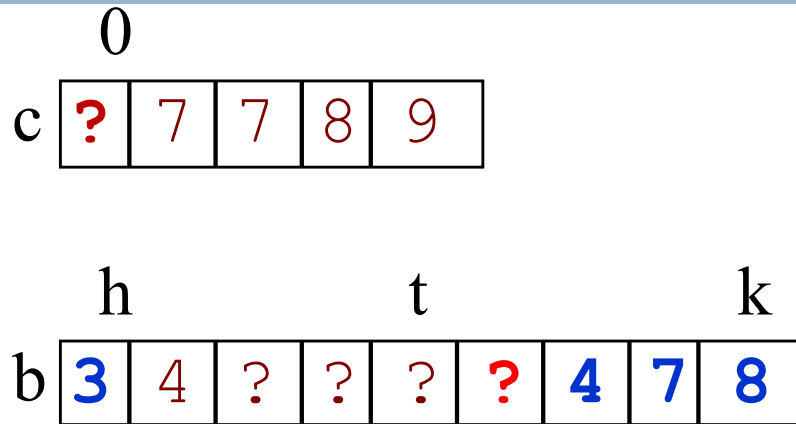
Step 2. Move 4
from c[0] to b[h+1]

h       t       k

b | 3 | 4 | ? | ? | ? | ? | 4 | 7 | 8 |

# Merge two adjacent sorted segments

0

c | ? | 7 | 7 | 8 | 9 |

h        t       k

b | 3 | 4 | 4 | ? | ? | ? | ? | 7 | 8 |
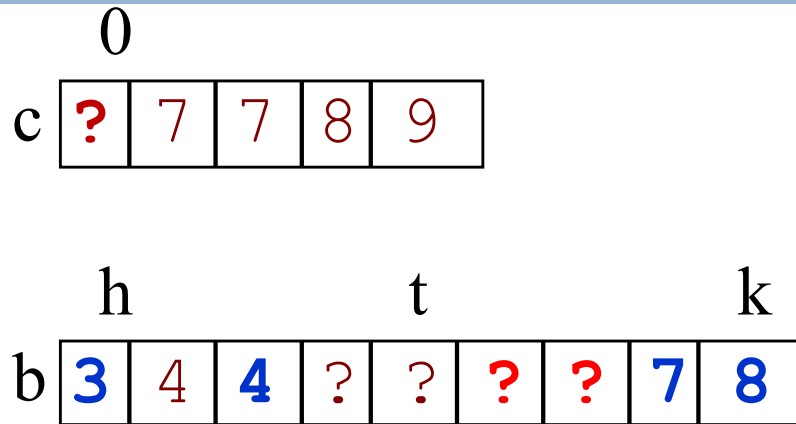
Step 3. Move **4**
from $b[t+2]$ to $b[h+2]$

# Merge two adjacent sorted segments

0

c | ? | ? | 7 | 7 | 8 | 9
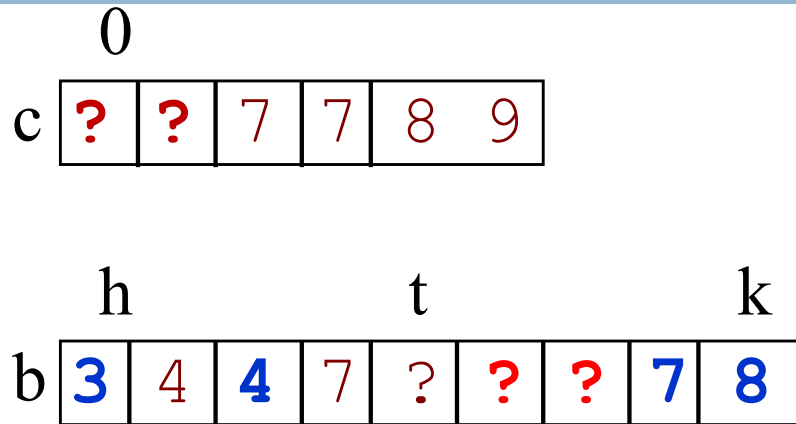
Step 4. Move 7
from c[1] to b[h+3]

h      t      k

b | 3 | 4 | 4 | 7 | ? | ? | ? | 7 | 8

if i < c.length and h < u,
b[u-1] ≤ c[i]

Invariant:

0      i

c | **moved** | still to move

if v ≤ k and h < u,
b[u-1] ≤ b[v]

h      u      v      k

b | **In place, sorted** | **?** | still to move
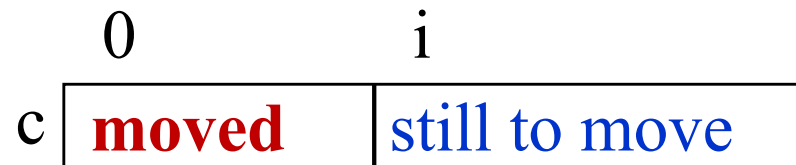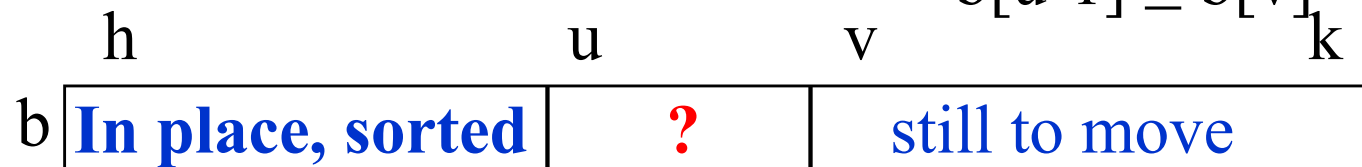
# Merge two adjacent sorted segments

// Merge sorted c and b[t+1..k] into b[h..k]

```
              0        t-h              h        t          k
pre:      c  [      x      ]        b  [    ?    |     y    ]      x, y are sorted
```

```
              h                     k
post: b  [      x and y, sorted      ]
```

```
              0              i           c.length
invariant:
          c  [  head of x  |  tail of x  ]
```

```
       h              u           v        k
   b  [              |     ?     | tail of  y ]
                     ↑
```

head of x and head of y, sorted

# Merge

```
i = 0;  u = h;  v = t+1;
while( i <= t-h){
    if(v <= k && b[v] < c[i]) {
        b[u] = b[v];
        u++; v++;
    }else {
        b[u] = c[i];
        u++; i++;
    }
}
}
```

pre: c [ sorted ]   b [ ? | sorted ]

0        t-h    h       t           k

post: b [ sorted ]

h                              k

inv:

0                    i                c.length

c [ sorted | sorted ]

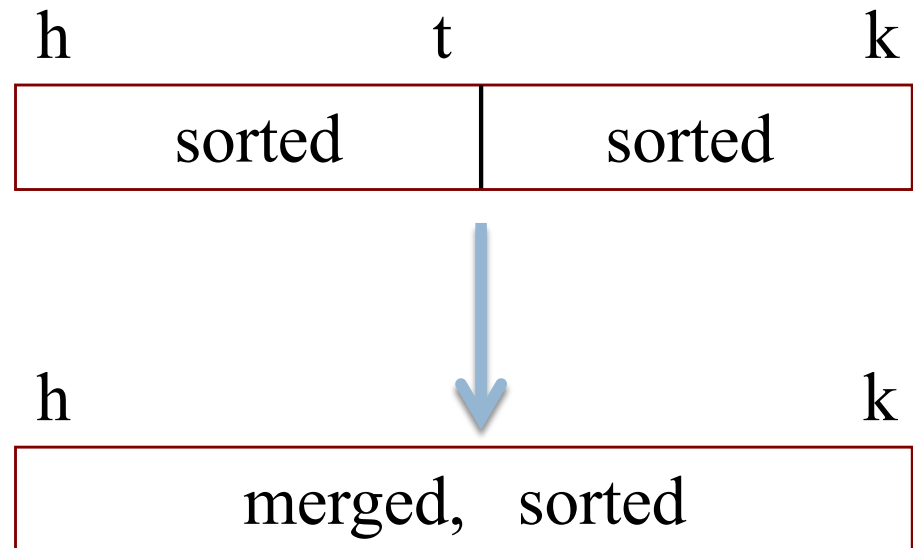h           u          v          k

b [ sorted | ? | sorted ]

# Mergesort

```
/** Sort b[h..k] */
public static void mergesort(int[] b, int h, int k]) {
    if (size b[h..k] < 2)  return;
    int t= (h+k)/2;
    mergesort(b, h, t);
    mergesort(b, t+1, k);
    merge(b, h, t, k);
}
```

| h | | t | | k |
|---|---|---|---|---|
| | sorted | | sorted | |

| h | | | | k |
|---|---|---|---|---|
| | | merged,   sorted | | |

# QuickSort versus MergeSort

```
/** Sort b[h..k] */
public static void QS
        (int[] b, int h, int k) {
    if (k – h < 1) return;
    int j= partition(b, h, k);
    QS(b, h, j-1);
    QS(b, j+1, k);
}
```

```
/** Sort b[h..k] */
public static void MS
        (int[] b, int h, int k) {
    if (k – h < 1) return;
    MS(b, h, (h+k)/2);
    MS(b, (h+k)/2 + 1, k);
    merge(b, h, (h+k)/2, k);
}
```

One processes the array then recurses.
One recurses then processes the array.

# Performance

| Algorithm | Ave time.   Worst-case time | | Space | Stable? |
|---|---|---|---|---|
| Insertion sort | $O(n^2).$ | $O(n^2)$ | $O(1)$ | Yes |
| Selection sort | $O(n^2).$ | $O(n^2)$ | $O(1)$ | No |
| Quick sort | $O(n \log n).$ | $O(n^2)$ | O(log n)* | No |
| Merge sort | $O(n \log n).$ | $O(n \log n)$ | O(n) | Yes |

\* The first algorithm we developed takes space O(n)
in the worst case, but it can be reduced to O(log n)

# Sorting in Java

- Java.util.Arrays has a method sort(array)
  - implemented as a collection of overloaded methods
  - for primitives, sort is implemented with a version of quicksort
  - for Objects that implement Comparable, sort is implemented with timSort, a modified mergesort developed in 1993 by Tim Peters
- Tradeoff between speed/space and stability/performance guarantees

# Quicksort with logarithmic space

Problem is that if the pivot value is always the smallest (or always the largest), the depth of recursion is the size of the array to sort.

Eliminate this problem by doing some of it iteratively and some recursively. We may show you this later. Not today!

# QuickSort with logarithmic space

```
/** Sort b[h..k]. */
public static void QS(int[] b, int h, int k) {
    int h1= h; int k1= k;
    // invariant b[h..k] is sorted if b[h1..k1] is sorted
    while (b[h1..k1] has more than 1 element) {
        Reduce the size of b[h1..k1], keeping inv true
    }
}
```

# QuickSort with logarithmic space

```
/** Sort b[h..k]. */

public static void QS(int[] b, int h, int k) {
    int h1= h; int k1= k;
    // invariant b[h..k] is sorted if b[h1..k1] is sorted
    while (b[h1..k1] has more than 1 element) {
        int j= partition(b, h1, k1);
        // b[h1..j-1] <= b[j] <= b[j+1..k1]
        if (b[h1..j-1] smaller than b[j+1..k1])
            {  QS(b, h, j-1);  h1= j+1; }
        else
            {QS(b, j+1, k1);  k1= j-1; }
    }
}
```

Only the smaller segment is sorted recursively. If b[h1..k1] has size n, the smaller segment has size < n/2. Therefore, depth of recursion is at most log n