"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

Lecture 11

CS2110 – Fall 2017

Prelim 1

- It's on Tuesday Evening (3/13)
- Two Sessions:
  - 5:30-7:00PM: netid aa..ks
  - 7:30-9:00PM: netid kt..zz
  - If you have a conflict with your assigned time but can make the other time, fill out conflict assignment on CMS BY TOMORROW
- Three Rooms:
  - We will email you Tuesday morning with your room
- Bring your Cornell ID!!!

Prelim 1

- Recitation 5: prelim review
- Review Session: Sunday 3/11, 1-3pm in Kimball B11
- Study guide on course website

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 about several different algorithms (and how to derive them)

Some Sorting Algorithms

- Insertion sort
- Selection sort
- Merge sort
- Quick 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

A loop that processes elements of an array in increasing order has this invariant

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:

	0	i	b.length
b	sorted	?	

e.g.

	0	i	b.length
b	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]}
```

Let n = b.length

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

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

Swaps.

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

Performance

Algorithm	Time	Space	Stable?
Insertion Sort	$O(n)$ to $O(n^2)$	$O(1)$	Yes
Selection Sort	$O(n^2)$	$O(1)$	No
Merge Sort			
Quick Sort			

SelectionSort

	0		b.length
pre: b	?		

	0		b.length
post: b	sorted		

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

Additional term in invariant

Keep invariant true while making progress?

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

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

SelectionSort

```
//sort b[], an array of int
// 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 minimum 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	

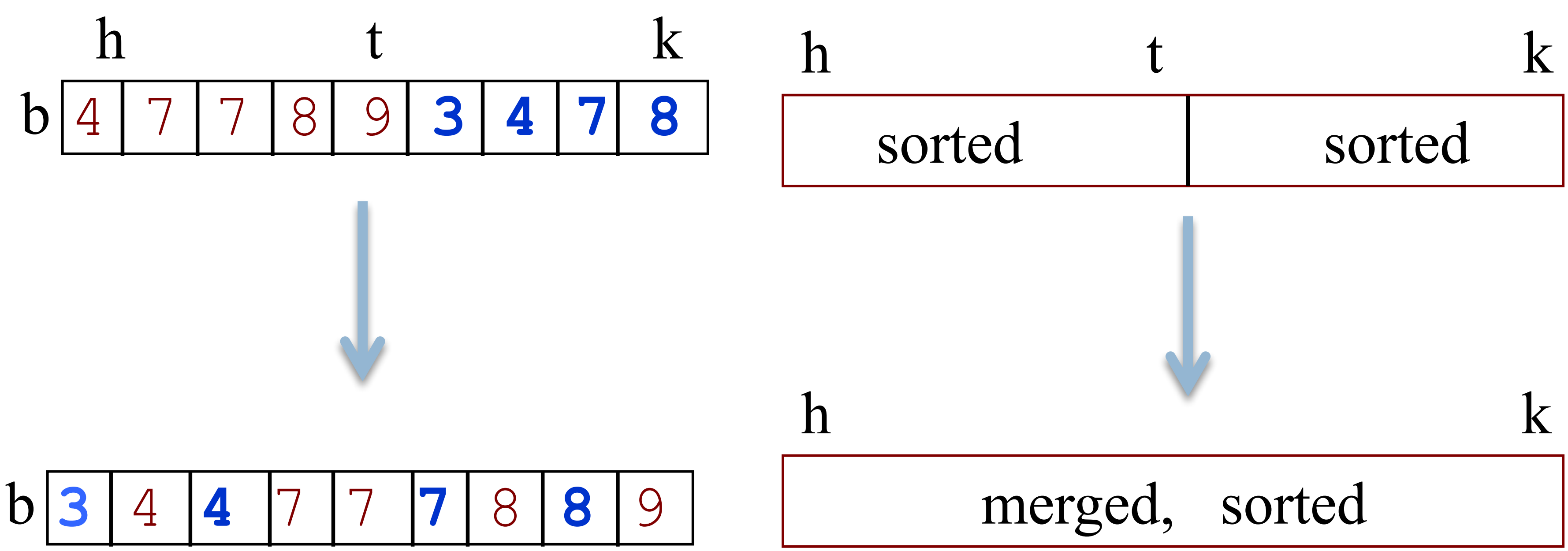
Each iteration, swap min value of this section into b[i]

Performance

Algorithm	Time	Space	Stable?
Insertion Sort	$O(n)$ to $O(n^2)$	$O(1)$	Yes
Selection Sort	$O(n^2)$	$O(1)$	No
Merge Sort			
Quick Sort			

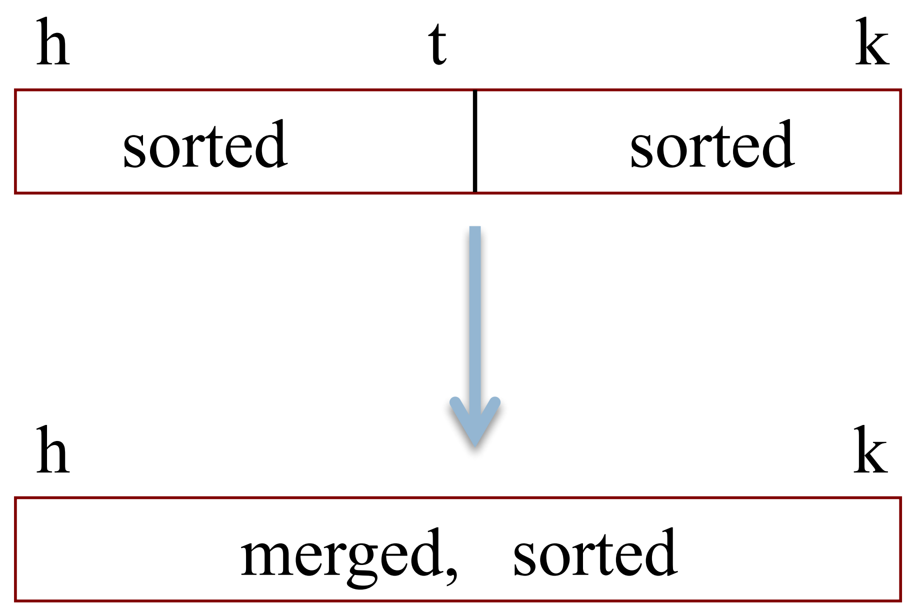
Merge two adjacent sorted segments

```
/* 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) {
}
```



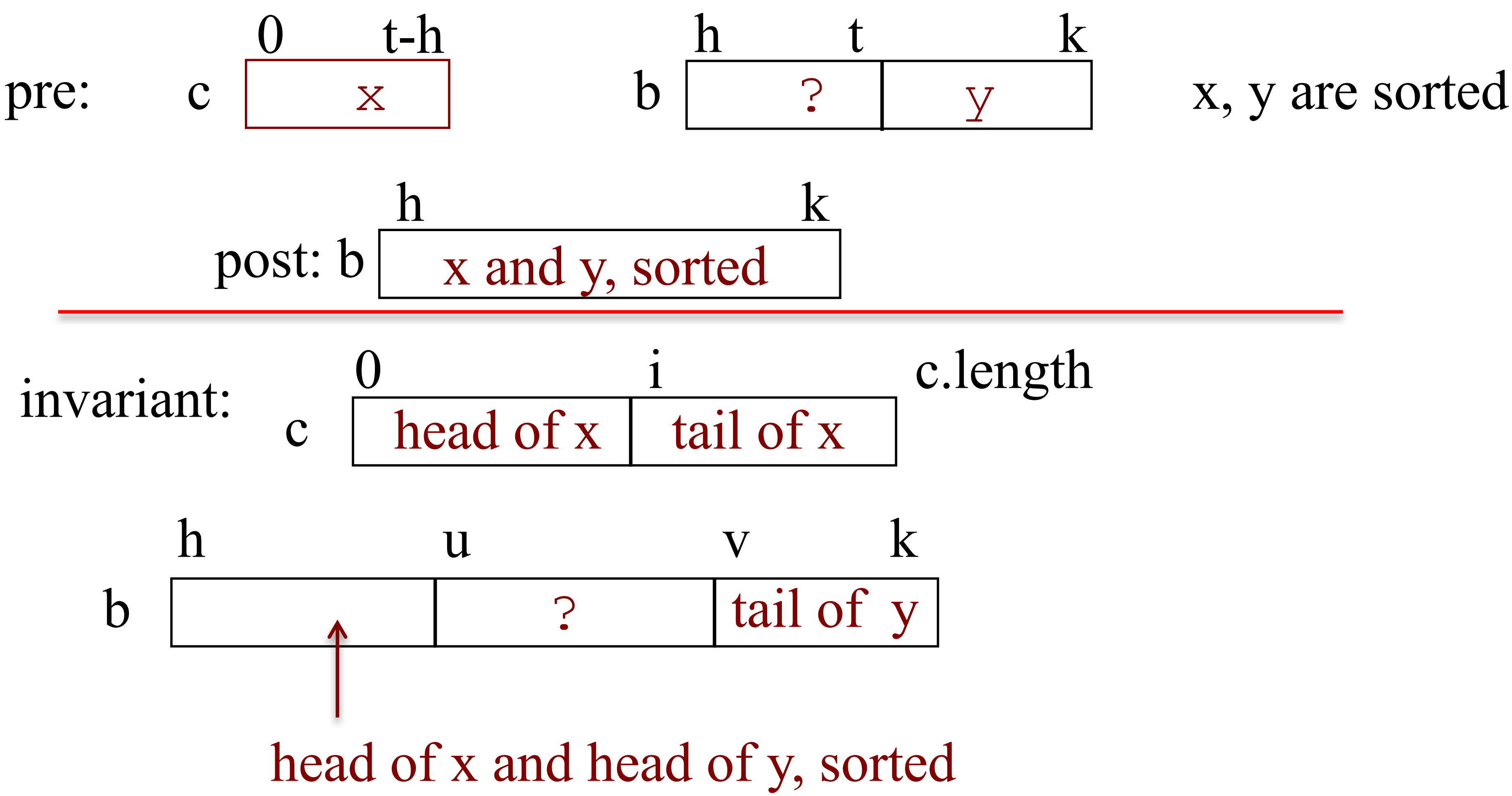
Merge two adjacent sorted segments


```
/* 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];
}
```



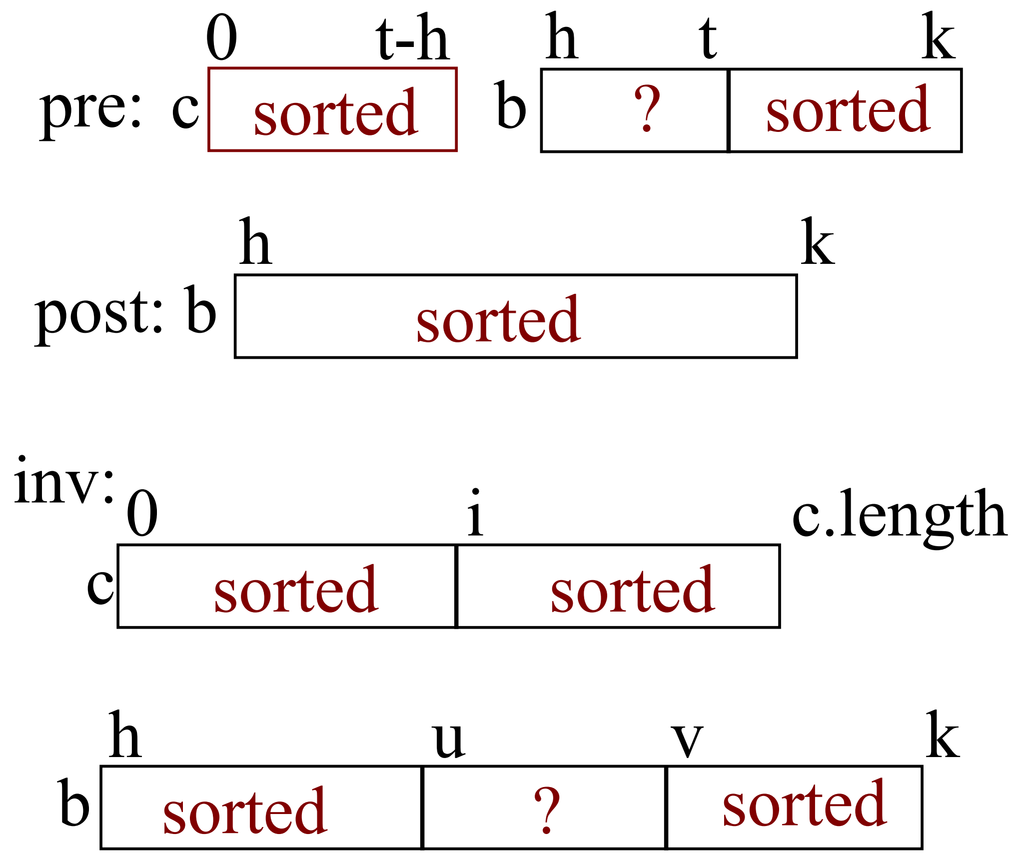
Merge two adjacent sorted segments

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



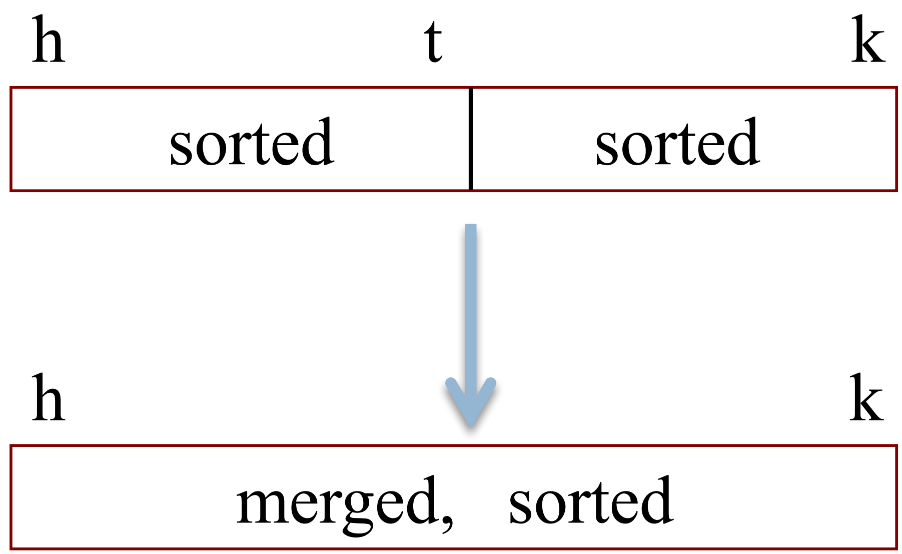
Merge

```
int i = 0;
int u = h;
int 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++;
      }
   }
}
```



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);
}
```



Performance

Algorithm	Time	Space	Stable?
Insertion Sort	$O(n)$ to $O(n^2)$	$O(1)$	Yes
Selection Sort	$O(n^2)$	$O(1)$	No
Merge Sort	$n\log(n)$	$O(n)$	Yes
Quick Sort			

QuickSort

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

83 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.

Partition algorithm of quicksort

h h+1 k

pre:

x	?

x is called the pivot

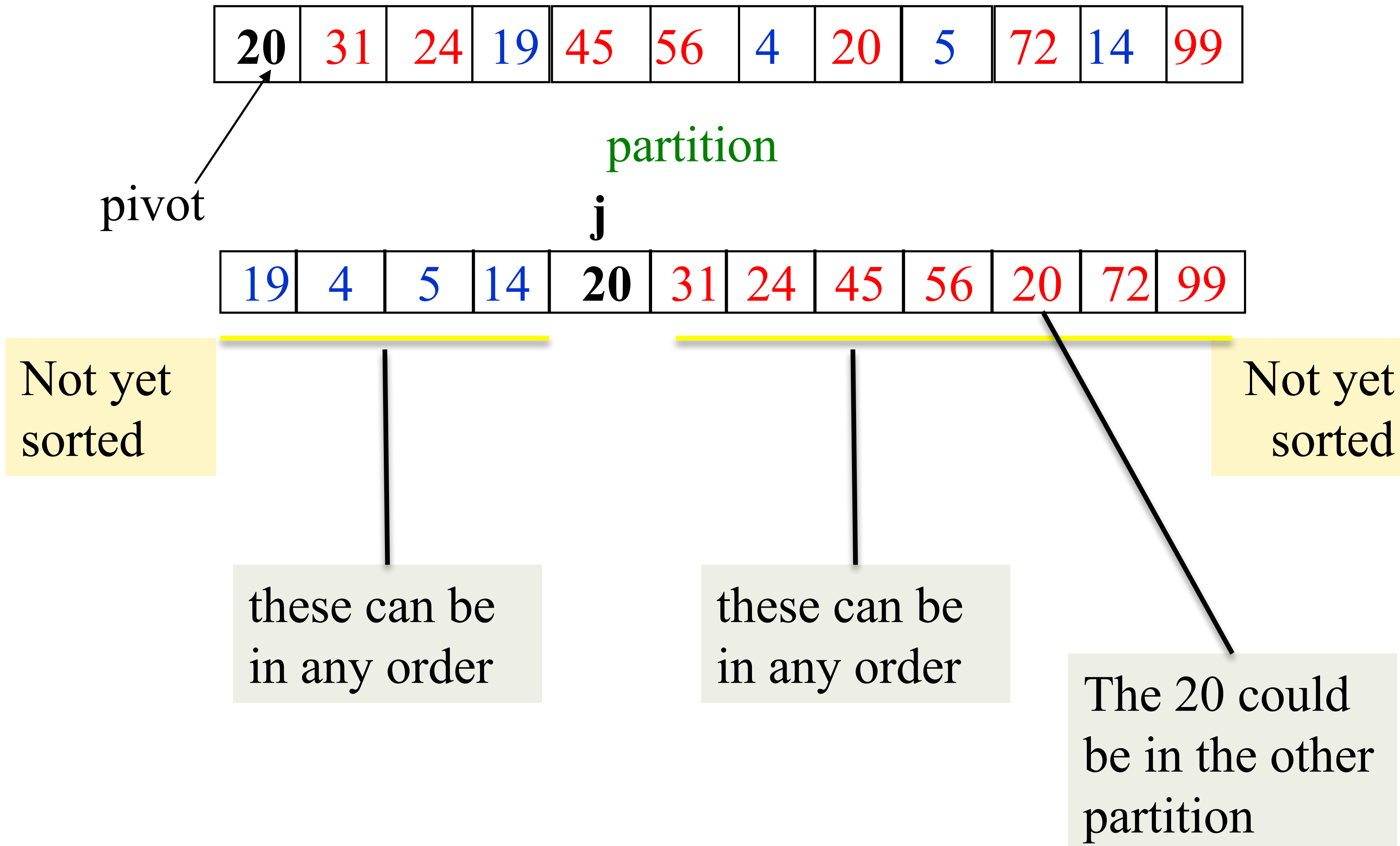
Swap array values around until b[h..k] looks like this:

h j k

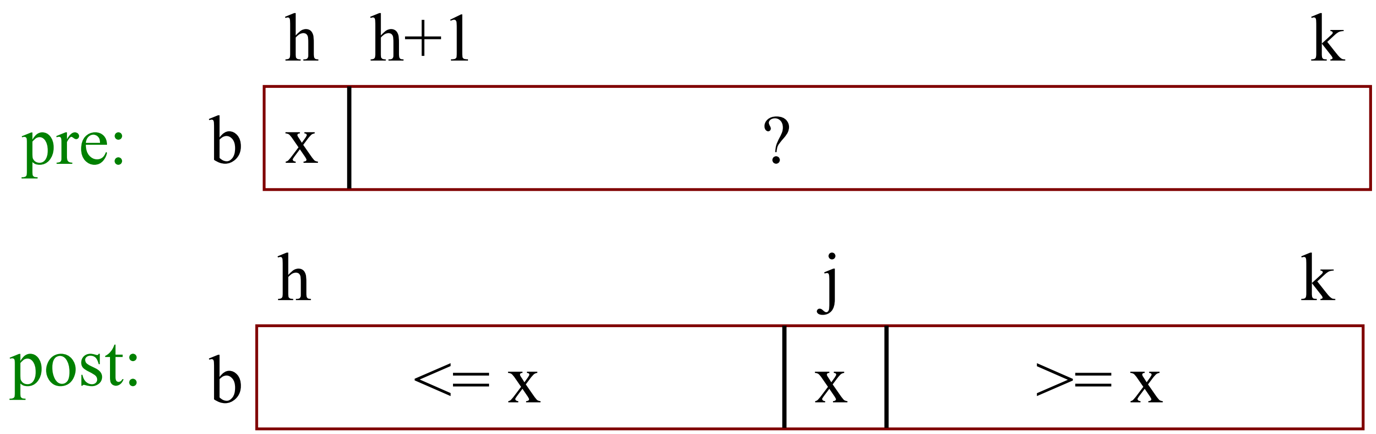
post:

<= x	x	>= x

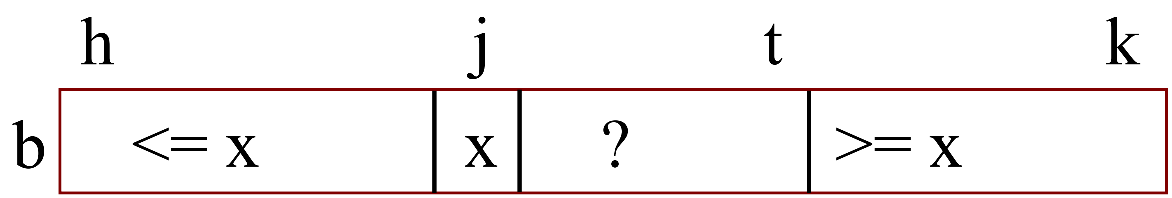
Partition algorithm of quicksort



Partition algorithm



Combine pre and post to get an invariant



invariant needs at least 4 sections

Partition algorithm

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

```
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;
    }
}
```

Initially, with j = h and t = k, this diagram looks like the start diagram

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

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

QuickSort procedure

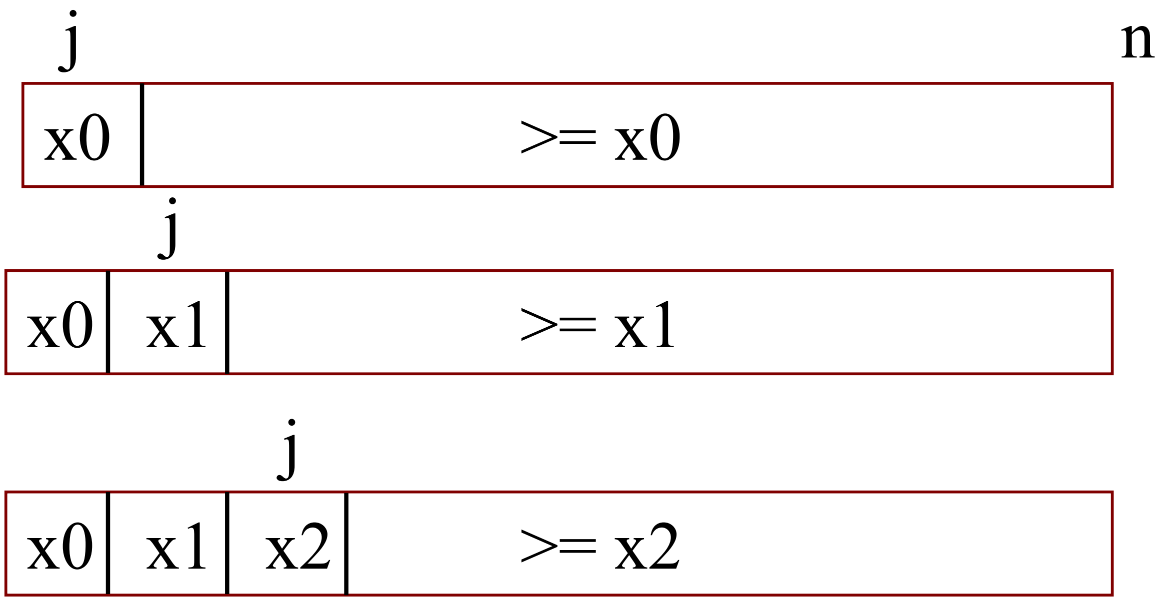
```
/** Sort b[h..k]. */
public static void QS(int[] b, int h, int k) {
    if (b[h..k] has < 2 elements) return;

    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);
}
```

Base case

Function does the partition algorithm and returns position j of pivot

h	j	k
<= x	x	>= x

Worst case quicksort: pivot always smallest value



partioning at depth 0

partioning at depth 1

partioning at depth 2

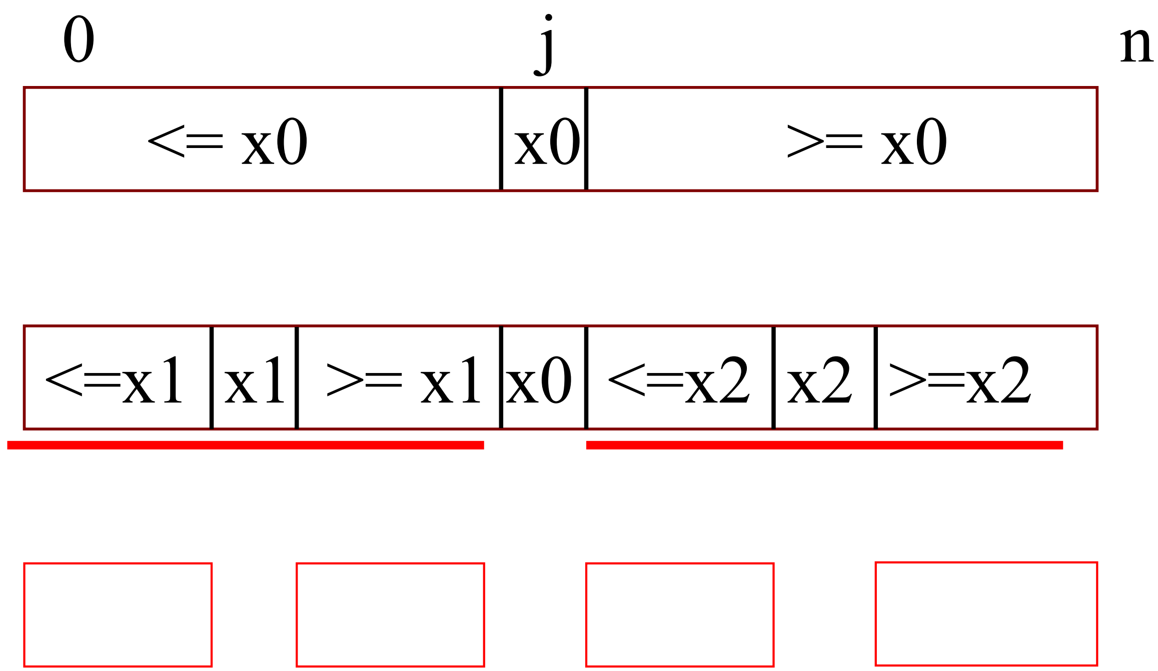
```
/** Sort b[h..k]. */
public static void QS(int[] b, int h, int k) {
    if (b[h..k] has < 2 elements) return;
    int j=  partition(b, h, k);
    QS(b, h, j-1);     QS(b, j+1, k);
```

Depth of recursion: O(n)

Processing at depth i: O(n-i)

O(n*n)

Best case quicksort: pivot always middle value



depth 0. 1 segment of size ~n to partition.

Depth 2. 2 segments of size ~n/2 to partition.

Depth 3. 4 segments of size ~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

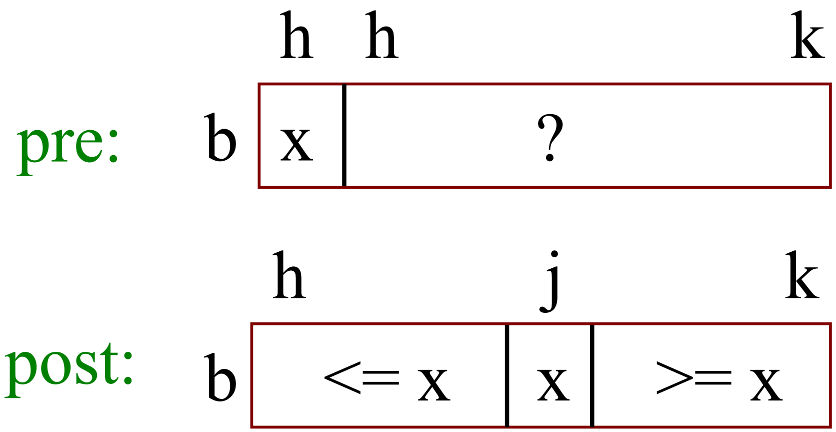
QuickSort versus MergeSort


```
/** 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.

Partition. Key issue. How to choose pivot



Choosing pivot

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

But computing 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	Time	Space	Stable?
Insertion Sort	$O(n)$ to $O(n^2)$	$O(1)$	Yes
Selection Sort	$O(n^2)$	$O(1)$	No
Merge Sort	$n\log(n)$	$O(n)$	Yes
Quick Sort	$n\log(n)$ to $O(n^2)$	$O(\log(n))$	No

Sorting in Java

- Java.util.Arrays has a method Sort()
  - 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 mergesort
- 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