COMSW 1003-1

Introduction to Computer Programming in C

Lecture 22

Spring 2011

Instructor: Michele Merler

http://www1.cs.columbia.edu/~mmerler/comsw1003-1.html

Today

- Quicksort
- Pointers to functions, implementation of `qsort()`
- HW4 solution

Review – Bubble Sort

1. Start with the first two elements
2. If first element > second element
    - Swap
3. Iterate for all following pairs
4. Repeat steps 1 to 3 until no swaps are necessary

Complexity = $O(n^2)$

Count number of comparisons and swaps

Review - Selection Sort

- Smarter algorithm, but same complexity (worst case)

1. Find smallest unsorted element
2. Swap with first unsorted element
3. Repeat steps 1 and 2 until no more unsorted elements

Complexity = $O(n^2)$

Review - Merge Sort

- One of the fastest algorithms, divide and conquer principle
- Uses recursion
- Sorting small sets is faster than sorting large sets
- Merging 2 sets into a sorted union is faster if the sets are already sorted

1. If set H has 1 element, stop
2. else
   - Split set into 2 halves H1 and H2 of (approximately) same size
   - Sort H1 and H2 with merge sort — recursion
   - Merge the sorted H1 and H2 into a sorted set

Complexity = O(n log(n))

Review - Counting sort

- Intuition: exploit range *k* of values in set
- Efficient if *k* is not much larger than *n*

1. Find biggest and smallest values in the set
   (k = maxVal – minVal+1)
2. Create an array C of k elements
3. Count occurrences *C(i)* of each value *i* in the set
4. Fill ordered set by inserting *C(i)* elements of value *i*, for each value in range *k*

Complexity = O(n + k)

Quicksort

- Divide and conquer idea (similar to merge sort)
- **In real world cases, on average it is as fast or faster than O(n log(n)) algorithms**

Complexity = O(n^2)

1. Choose an element in the array called **pivot P** and remove it from the array (common choice is median of first, middle and last element)

2. For each element *x* in the array (minus pivot)

   if(x < pivot)

      insert x in set *S* of elements smaller than pivot

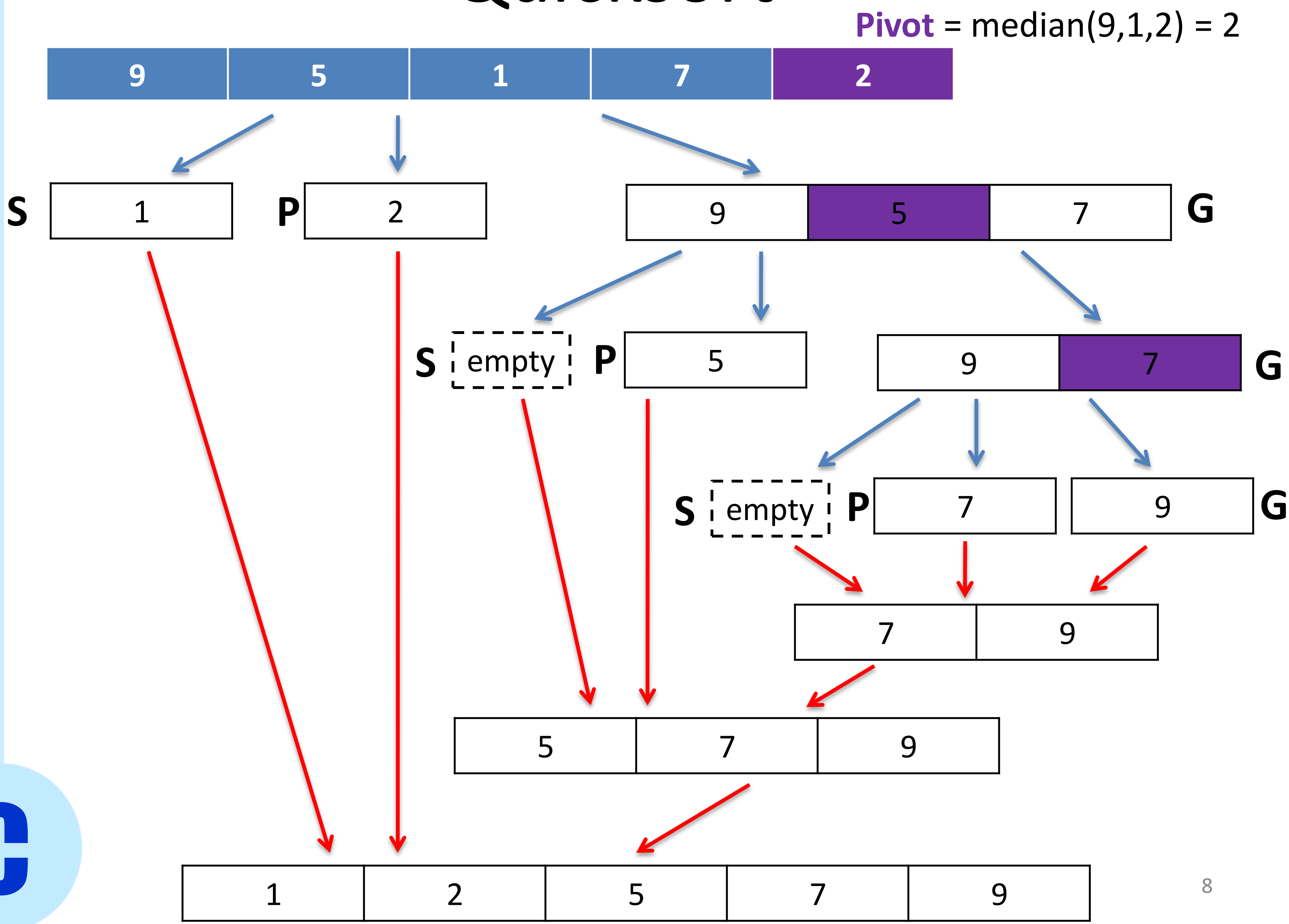
   else

      insert x in set *G* of elements greater than pivot

3. return(concatenate(quicksort(S), P , quicksort(G)));

Recursive call!

Quicksort


Pivot = median(9,1,2) = 2
9 5 1 7 2
S 1
P 2
9 5 7 G
S empty
P 5
9 7 G
S empty
P 7
9 G
7 9
5 7 9
1 2 5 7 9

Pointers to functions

Pointers to functions

- It is occasionally useful to use pointers to functions
- Since functions are stored in memory, we can reason about their addresses too
- This allows us to say, “run the function at address *N* on these arguments”
- Useful for being truly general, e.g. stdlib qsort

Pointers to functions

```
int (*f_ptr)(); // pointer to function that returns an int
```

Parentheses are important! Without parentheses, **f_ptr looks like it returns a** pointer to an int.

```
int (*f_ptr)(int, int);  // pointer to a function

int greater_than(int a, int b);  // function declaration

f_ptr = greater_than;
```

Pointers to functions

```
int (*f_ptr)(); // pointer to function that returns an int
```

Parentheses are important! Without parentheses, **f_ptr looks like it returns a** pointer to an int.

```
int (*f_ptr)(int, int);

int greater_than(int a, int b);

f_ptr = greater_than;
```

```
int *ptr;

int x[2];

ptr = x;
```

qsort

- qsort() is a general sorting function, defined in stdlib.h
- Sort an array of any type, using any comparison criterion
- Define that comparison as a function pointer

```
void qsort(void *base, size_t n, size_t size,
           int (*cmp)(const void *, const void *));
```

Depending on what function `cmp` points to, qsort uses a different criterion to sort the data

qsort

sorting.c

```
void qsort(void *base, size_t n, size_t size,
           int (*cmp)(const void *, const void *));
```

Depending on what function `cmp` points to, qsort uses a different criterion to sort the data

The compare function should take two entries *x* and *y*, and return

+1	if	x > y
-1	if	x < y
0	if	x == y