Bubble Sort and Quick-sort

EECS 233

Previous Lecture

- Method 1: Selection Sort
  - For each position of an array, find the element for it
  - $O(n^2)$ comparisons, $O(n)$ moves

0	1	2	3	4	5	6
2	4	7	21	25	10	17

- Method 2: Insertion Sort
  - For each element, find a position to insert it
  - $O(n^2)$ comparisons, $O(n^2)$ moves
  - O(n) if the array is sorted or nearly sorted

0	1	2	3	4
6	14	19	9	...

- Method 3: Shell Sort
  - Based on the observation that insertion sort requires O(n) running time for sorted or nearly sorted array
  - Generalization of insertion sort with larger “jumps”, using strides larger than 1 (for insertion sort, the stride = 1)
  - Use a decreasing sequence of strides

0	1	2	3	4	5	6	7	8
99	8	13	2	15	9	4	12	24

Method 4: Bubble Sort

- Perform a sequence of passes through the array.
- On each pass: proceed from left to right, swapping adjacent elements if they are out of order. Larger elements “bubble up” to the end of the array.
- At the end of the kth pass, the k rightmost elements are in their final positions.
- Example:

	0	1	2	3
	28	24	27	18
After one pass	24	27	18	*28*
After two passes	24	18	*27*	*28*
After three passes	18	*24*	*27*	*28*

Implementation of Bubble Sort

0	1	2	3
28	24	27	18
24	27	18	*28*
24	18	*27*	*28*
18	*24*	*27*	*28*

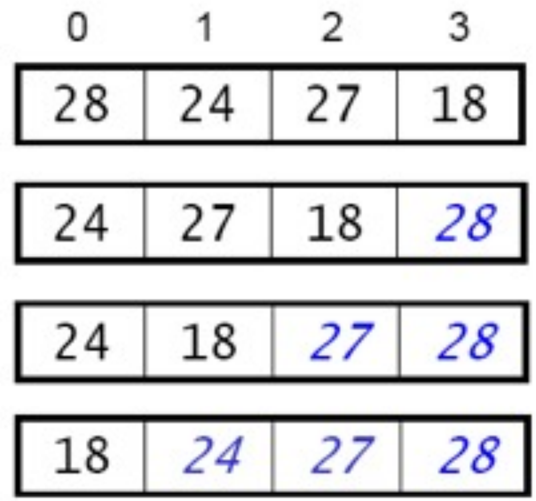
Implementation of Bubble Sort

static void bubbleSort(int[] arr, int length) {

0	1	2	3
28	24	27	18
24	27	18	*28*
24	18	*27*	*28*
18	*24*	*27*	*28*

Implementation of Bubble Sort

```
static void bubbleSort(int[] arr, int length) {
    for (int i = length – 1; i > 0; i--) {
        for (int j = 0; j < i; j++) {
            if (arr[j] > arr[j+1])
                swap(arr, j, j+1);
        }
    }
}
```



Implementation of Bubble Sort

```
static void bubbleSort(int[] arr, int length) {
      for (int i = length – 1; i > 0; i--) {
            for (int j = 0; j < i; j++) {
                  if (arr[j] > arr[j+1])
                     swap(arr, j, j+1);
            }
      }
}
```

0	1	2	3
28	24	27	18
24	27	18	*28*
24	18	*27*	*28*
18	*24*	*27*	*28*

- The inner loop performs a single pass

Implementation of Bubble Sort

```
static void bubbleSort(int[] arr, int length) {
    for (int i = length – 1; i > 0; i--) {
        for (int j = 0; j < i; j++) {
            if (arr[j] > arr[j+1])
                swap(arr, j, j+1);
        }
    }
}
```

0	1	2	3
28	24	27	18
24	27	18	*28*
24	18	*27*	*28*
18	*24*	*27*	*28*

- The inner loop performs a single pass
- The outer loop governs the number of passes, and the ending point of each pass

Running Time Analysis

```
static void bubbleSort(int[] arr, int n) {
        for (int i = n – 1; i > 0; i--) {
                for (int j = 0; j < i; j++) {
                        if (arr[j] > arr[j+1])
                            swap(arr, j, j+1);
                }
        }
}
```

0	1	2	3
28	24	27	18
24	27	18	*28*
24	18	*27*	*28*
18	*24*	*27*	*28*

- Number of comparisons
  - the k-th pass performs ? comparisons, k=1,2,…,n-1
  - so we get C(n) = ?
- Moves: depends on the contents of the array
  - in the worst case: the array is in reverse order, and every comparison leads to a swap (3 moves), so M(n) = ?
  - in the best case: the array is already sorted, and no moves are needed
  - Average: 50% chance a comparison leads to a swap
- Total running time: ?
  - C(n) is always $O(n^2)$, M(n) is never worse than $O(n^2)$.

Running Time Analysis

```
static void bubbleSort(int[] arr, int n) {
        for (int i = n – 1; i > 0; i--) {
                for (int j = 0; j < i; j++) {
                        if (arr[j] > arr[j+1])
                            swap(arr, j, j+1);
                }
        }
}
```

0	1	2	3
28	24	27	18
24	27	18	*28*
24	18	*27*	*28*
18	*24*	*27*	*28*

- Number of comparisons
  - the k-th pass performs n-k comparisons, k=1,2,…,n-1
  - so we get C(n) = ?
- Moves: depends on the contents of the array
  - in the worst case: the array is in reverse order, and every comparison leads to a swap (3 moves), so M(n) = ?
  - in the best case: the array is already sorted, and no moves are needed
  - Average: 50% chance a comparison leads to a swap
- Total running time: ?
  - C(n) is always $O(n^2)$, M(n) is never worse than $O(n^2)$.

Running Time Analysis

```
static void bubbleSort(int[] arr, int n) {
        for (int i = n – 1; i > 0; i--) {
                for (int j = 0; j < i; j++) {
                        if (arr[j] > arr[j+1])
                            swap(arr, j, j+1);
                }
        }
}
```

0	1	2	3
28	24	27	18
24	27	18	*28*
24	18	*27*	*28*
18	*24*	*27*	*28*

- Number of comparisons
  - the k-th pass performs $n-k$ comparisons, k=1,2,…,n-1
  - so we get C(n) = $(n-1) + (n-2) + \ldots + 1 = n^2/2-n/2 = O(n^2)$
- Moves: depends on the contents of the array
  - in the worst case: the array is in reverse order, and every comparison leads to a swap (3 moves), so M(n) = ?
  - in the best case: the array is already sorted, and no moves are needed
  - Average: 50% chance a comparison leads to a swap
- Total running time: ?
  - C(n) is always $O(n^2)$, M(n) is never worse than $O(n^2)$.

Running Time Analysis

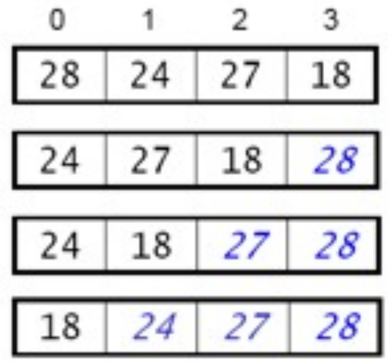
```
static void bubbleSort(int[] arr, int n) {
        for (int i = n – 1; i > 0; i--) {
                for (int j = 0; j < i; j++) {
                        if (arr[j] > arr[j+1])
                            swap(arr, j, j+1);
                }
        }
}
```

0	1	2	3
28	24	27	18
24	27	18	*28*
24	18	*27*	*28*
18	*24*	*27*	*28*

- Number of comparisons
  - the k-th pass performs n-k comparisons, k=1,2,…,n-1
  - so we get C(n) = $(n-1) + (n-2) + \dots + 1 = n^2/2 - n/2 = O(n^2)$
- Moves: depends on the contents of the array
  - in the worst case: the array is in reverse order, and every comparison leads to a swap (3 moves), so M(n) = $3C(n) = O(n^2)$
  - in the best case: the array is already sorted, and no moves are needed
  - Average: 50% chance a comparison leads to a swap
- Total running time: ?
  - C(n) is always $O(n^2)$, M(n) is never worse than $O(n^2)$.

Running Time Analysis

```
static void bubbleSort(int[] arr, int n) {
        for (int i = n – 1; i > 0; i--) {
                for (int j = 0; j < i; j++) {
                        if (arr[j] > arr[j+1])
                            swap(arr, j, j+1);
                }
        }
}
```



- Number of comparisons
  - the k-th pass performs n-k comparisons, k=1,2,…,n-1
  - so we get C(n) = $(n-1) + (n-2) + \ldots + 1 = n^2/2 - n/2 = O(n^2)$
- Moves: depends on the contents of the array
  - in the worst case: the array is in reverse order, and every comparison leads to a swap (3 moves), so M(n) = $3C(n) = O(n^2)$
  - in the best case: the array is already sorted, and no moves are needed
  - Average: 50% chance a comparison leads to a swap
- Total running time: $O(n^2)$
  - C(n) is always $O(n^2)$, M(n) is never worse than $O(n^2)$.

Method 5: Quick-Sort

- Like bubble sort, quick-sort uses an approach based on exchanging out-of-place elements, but it's more efficient. Analogy:
  - Insertion sort to Shell sort: jump faster
  - Bubble sort to quick sort: bubble faster

- A divide-and-conquer method:
  - *divide:* rearrange the elements so that we end up with two sub-arrays such that: *each element in the left array <= each element in the right array*
    - ✓ example:

0	1	2	3	4	5	6	7	8
16	**8**	**13**	**2**	**15**	**9**	**4**	**12**	**24**

→

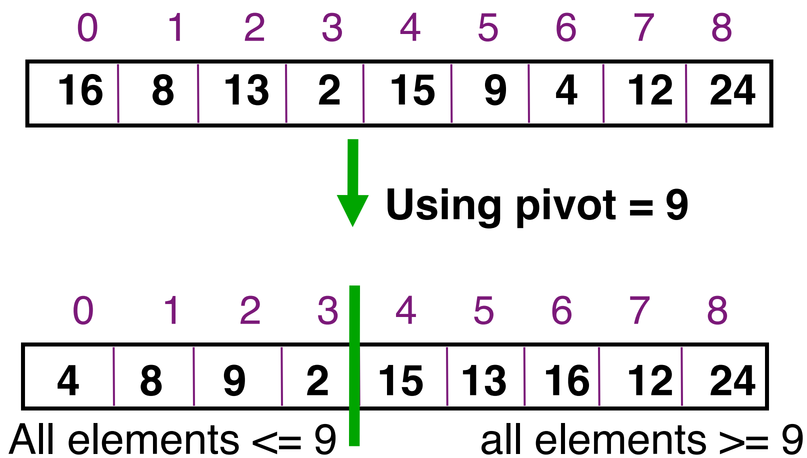
0	1	2	3	4	5	6	7	8
4	**8**	**9**	**2**	**15**	**13**	**16**	**12**	**24**

  - *conquer:* apply quick-sort recursively to the subarrays, stopping when a subarray has a single element
  - *combine:* nothing needs to be done, because of the criterion used in forming the subarrays

Partitioning An Array

- The process that quick-sort uses to rearrange the elements is known as *partitioning* the array.

- Partitioning is done using a value known as the *pivot.* We rearrange the elements to produce two subarrays:
  - left subarray: all values <= pivot
  - right subarray: all values >= pivot



An Example of Partitioning An Array

Pivot = middle element

	first							last
arr →	7	15	4	*9*	6	18	9	12

- Maintain indices i and j, starting them “outside” the array:

```
i = first - 1
j = last + 1
```

i									j
	7	15	4	9	6	18	9	12	

- *Find* “out-of-place” elements:
  - increment i until arr[i] >= pivot
  - decrement j until arr[j] <= pivot

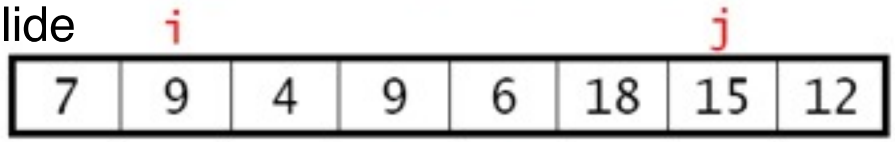
	i					j	
7	15	4	9	6	18	9	12

- *Swap* arr[i] and arr[j] if necessary

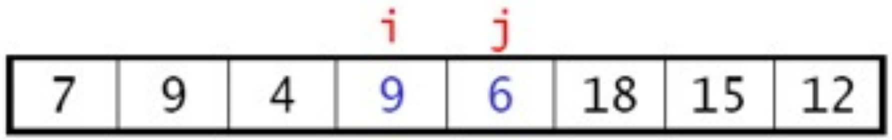
	i					j	
7	9	4	9	6	18	15	12

Partitioning An Array

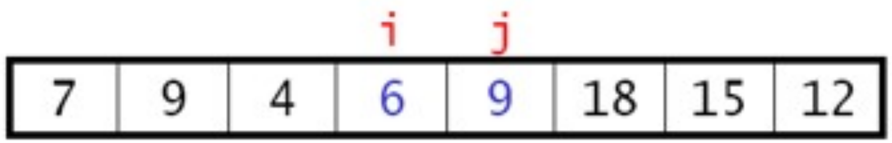
- From previous slide

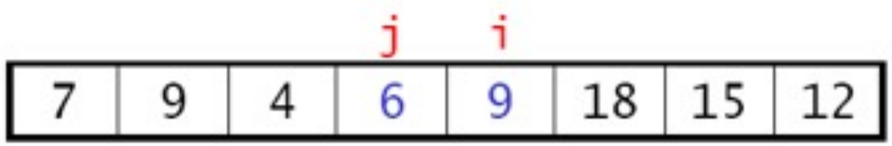


- Advance i and j



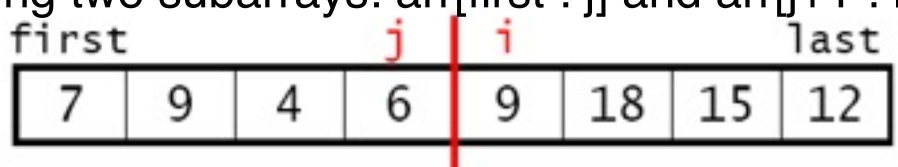
- Need to swap





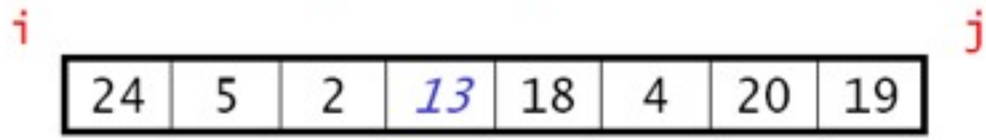
- i and j cross

- Return j, indicating two subarrays: arr[first : j] and arr[j+1 : last]

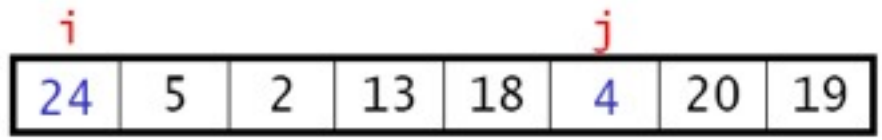


Another Example

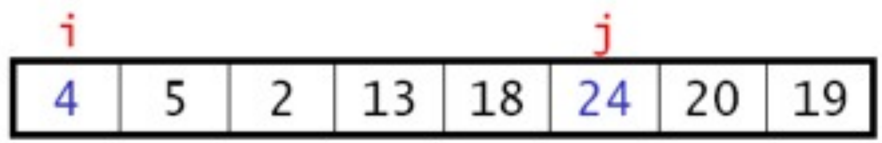
- Choose pivot=13



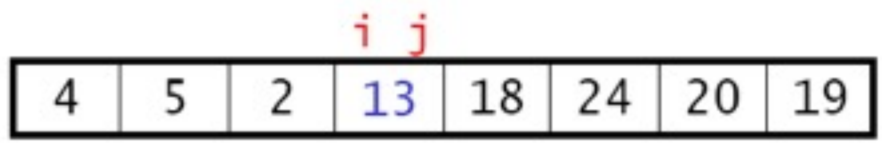
- Advance i and j



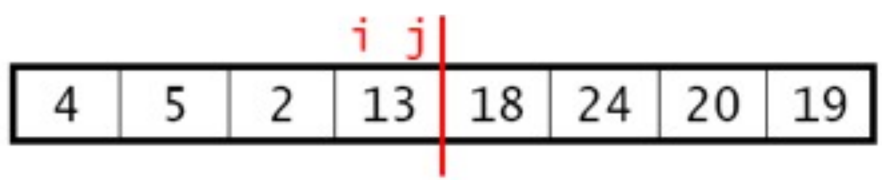
- Swap 4 and 24



- i and j meet



- Return j and we have two subarrays: arr[first : j] and arr[j+1 : last]



An exercise

i j

0	1	2	3	4	5	6	7	8
16	8	13	2	9	15	4	12	24

An exercise

i j

0	1	2	3	4	5	6	7	8
16	8	13	2	9	15	4	12	24

An exercise

i						j		
0	1	2	3	4	5	6	7	8
16	8	13	2	9	15	4	12	24

An exercise

i						j		
0	1	2	3	4	5	6	7	8
4	8	13	2	9	15	16	12	24

--

An exercise

	i					j		
0	1	2	3	4	5	6	7	8
4	8	13	2	9	15	16	12	24

An exercise

	i		j					
0	1	2	3	4	5	6	7	8
4	8	13	2	9	15	16	12	24

An exercise

	i		j					
0	1	2	3	4	5	6	7	8
4	2	13	8	9	15	16	12	24

An exercise

	i		j					
0	1	2	3	4	5	6	7	8
4	2	13	8	9	15	16	12	24

An exercise

		i	j					
0	1	2	3	4	5	6	7	8
4	2	13	8	9	15	16	12	24

An exercise

	j	i						
0	1	2	3	4	5	6	7	8
4	2	13	8	9	15	16	12	24

An exercise

	j	i						
0	1	2	3	4	5	6	7	8
4	2	13	8	9	15	16	12	24

Implementation of Quick-Sort

```
static void quickSort(int[] arr, int length) {
        qSort(arr, 0, length – 1);
}

static void qSort(int[] arr, int first, int last) {
        if (arr.length == 1) return;
        int split = partition(arr, first, last);



        ?


}
```

- quickSort() is the methods provided by the Sort class, which calls a recursive method
- The recursive method qSort() stops when the subarray size is 1

Implementation of Quick-Sort

```
static void quickSort(int[] arr, int length) {
    qSort(arr, 0, length – 1);
}

static void qSort(int[] arr, int first, int last) {
    if (arr.length == 1) return;
    int split = partition(arr, first, last);

    qSort(arr, first, split);
    qSort(arr, split+1, last);

}
```

- quickSort() is the methods provided by the Sort class, which calls a recursive method
- The recursive method qSort() stops when the subarray size is 1

Implementation of Quick-Sort

- The helper method partition()

```
static int partition(int[] arr, int first, int last)
{
        int pivot = arr[(first + last)/2];
        int i = first - 1;      // index going from left to right
        int j = last + 1;       // index going from right to left
        while (true) {



                ?









        }
}
```

Implementation of Quick-Sort

- The helper method partition()

```
static int partition(int[] arr, int first, int last)
{
    int pivot = arr[(first + last)/2];
    int i = first - 1;      // index going from left to right
    int j = last + 1;       // index going from right to left
    while (true) {
        do {
            i++;
        } while (arr[i] < pivot);
        do {
            j--;
        } while (arr[j] > pivot);
        if (i < j)
            swap(arr, i, j);
        else
            return j;  // arr[j] = end of left array
    }
}
```

Choosing Pivot Values (for Partitioning)

- First element or last element
  - risky, can lead to terrible worst-case behavior
  - especially poor if the array is almost sorted

0	1	2	3	4	5	6	7	8
16	**8**	**13**	**2**	**15**	**9**	**4**	**12**	**24**

→

0	1	2	3	4	5	6	7	8
16	**8**	**13**	**2**	**15**	**9**	**4**	**12**	**24**

- Middle element (good if the array is sorted)

- Randomly chosen element

- Median of three elements: to decrease the probability of getting a poor pivot
  - left, center, and right elements
  - three randomly selected elements

Running Time Analysis - Best Case

- Partitioning an array requires n comparisons, because each element is compared with the pivot.
- ***best case***:

Running Time Analysis - Best Case

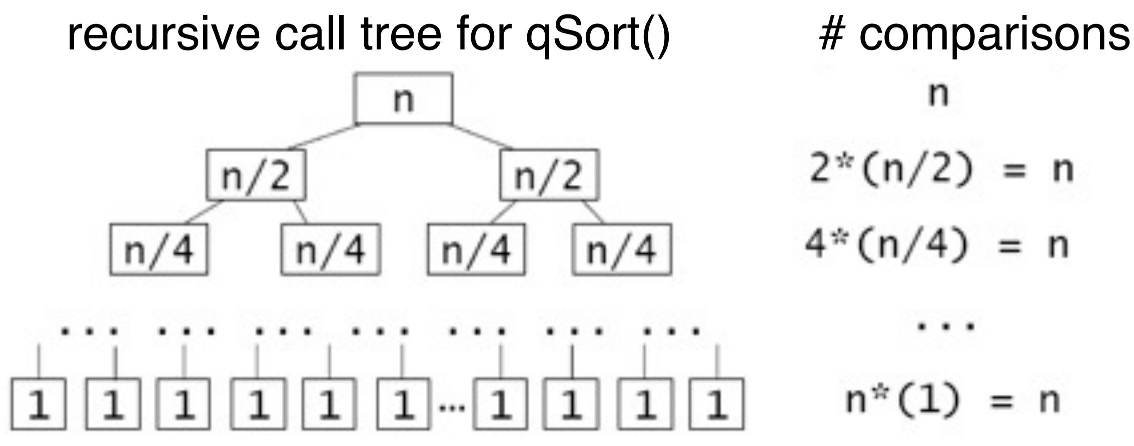
- Partitioning an array requires n comparisons, because each element is compared with the pivot.

- ***best case:*** partitioning always divides the array in half

Running Time Analysis - Best Case

- Partitioning an array requires n comparisons, because each element is compared with the pivot.

- ***best case:*** partitioning always divides the array in half



  - at each level of the call tree, we perform n comparisons
  - There are $\log_2 n$ levels in the tree. So $C(n) = n \log_2 n = O(n \log n)$
  - Similarly, $M(n)$ and running time are both $O(n \log n)$

Running Time Analysis - Worst Case

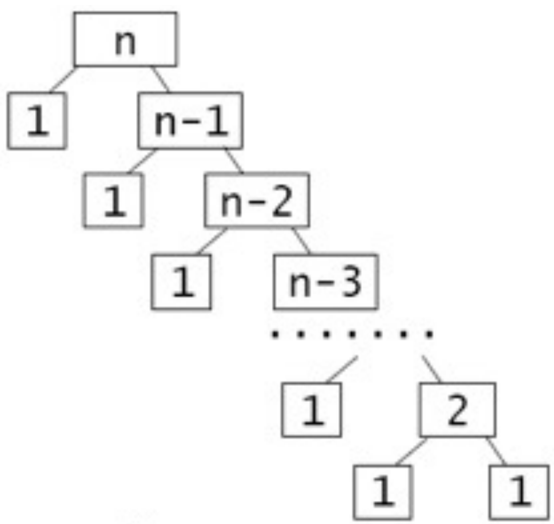
- ***Worst case***:

Running Time Analysis - Worst Case

- ***Worst case***: pivot is always the smallest or largest element one subarray has 1 element, the other has n - 1

Running Time Analysis - Worst Case

- ***Worst case***: pivot is always the smallest or largest element one subarray has 1 element, the other has n - 1
  - call tree for qSort



comparisons

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

  - C(n) is on the order of $O(n^2)$. What is M(n)?

- Average case: is harder to analyze. C(n) is still O(n log n)