CSE 373: Data Structures and Algorithms

Lecture 7: Sorting II

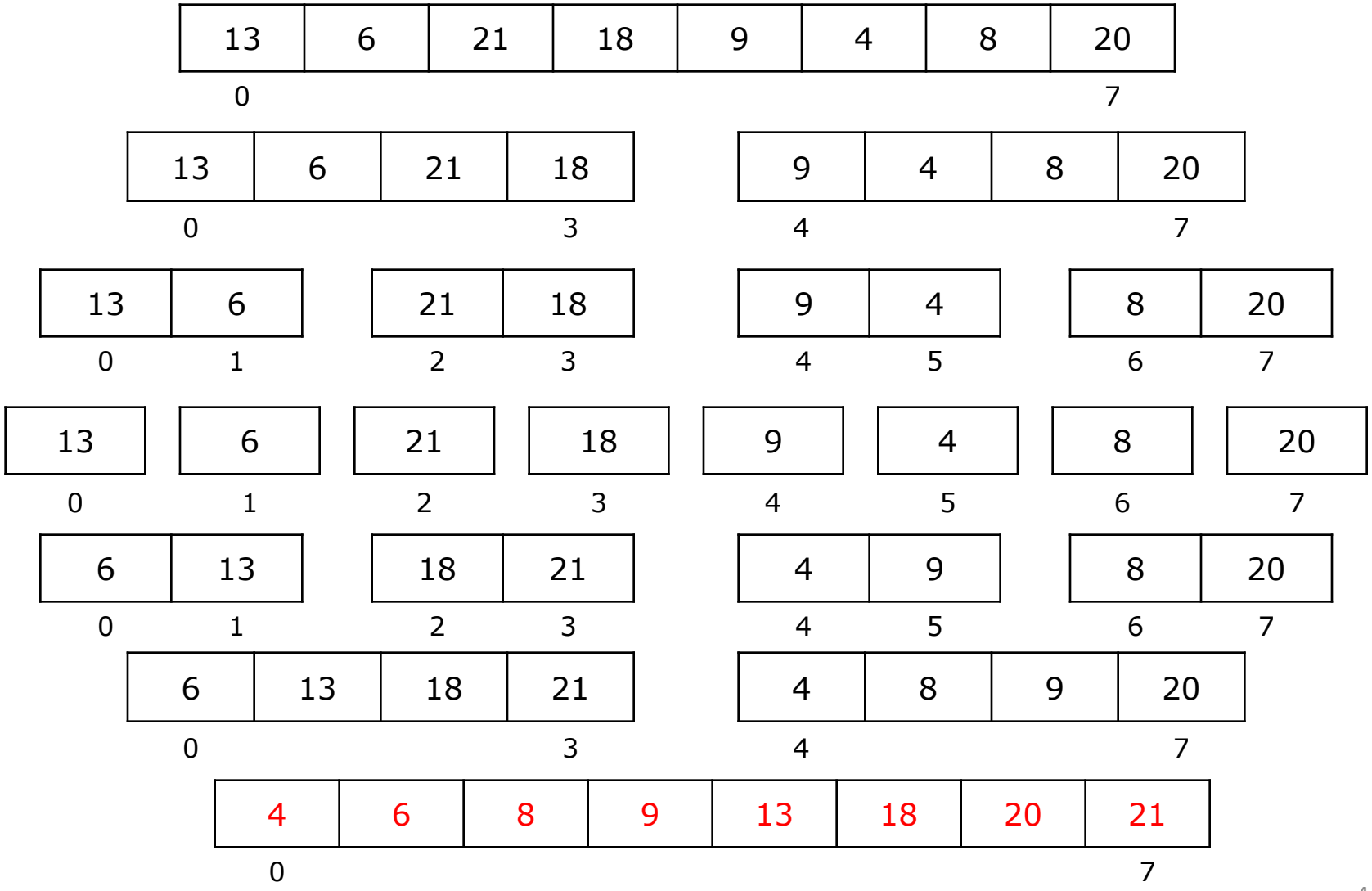
Sorting Classification

In memory sorting			External sorting
Comparison sorting $\Omega(N \log N)$		**Specialized Sorting**	
$O(N^2)$	**$O(N \log N)$**	**$O(N)$**	**# of disk accesses**
• Bubble Sort • Selection Sort • Insertion Sort • Shellsort Sort	• Merge Sort • Quick Sort • Heap Sort	• Bucket Sort • Radix Sort	• Simple External Merge Sort • Variations

in place? stable?

O(n log n) Comparison Sorting (continued)

Merge sort example 2



Quick sort

- **quick sort**: orders a list of values by partitioning the list around one element called a *pivot*, then sorting each partition
  - invented by British computer scientist C.A.R. Hoare in 1960

- more specifically:
  - choose one element in the list to be the pivot (partition element)
  - organize the elements so that all elements less than the pivot are to its left and all greater are to its right
  - apply the quick sort algorithm (recursively) to both partitions

Quick sort, continued

- For correctness, it's okay to choose any pivot.
- For efficiency, one of following is best case, the other worst case:
  - pivot partitions the list roughly in half
  - pivot is greatest or least element in list
- Which case above is best?
- We will divide the work into two methods:
  - `quickSort` – performs the recursive algorithm
  - `partition` – rearranges the elements into two partitions

Quick sort pseudo-code

- Let S be the input set.

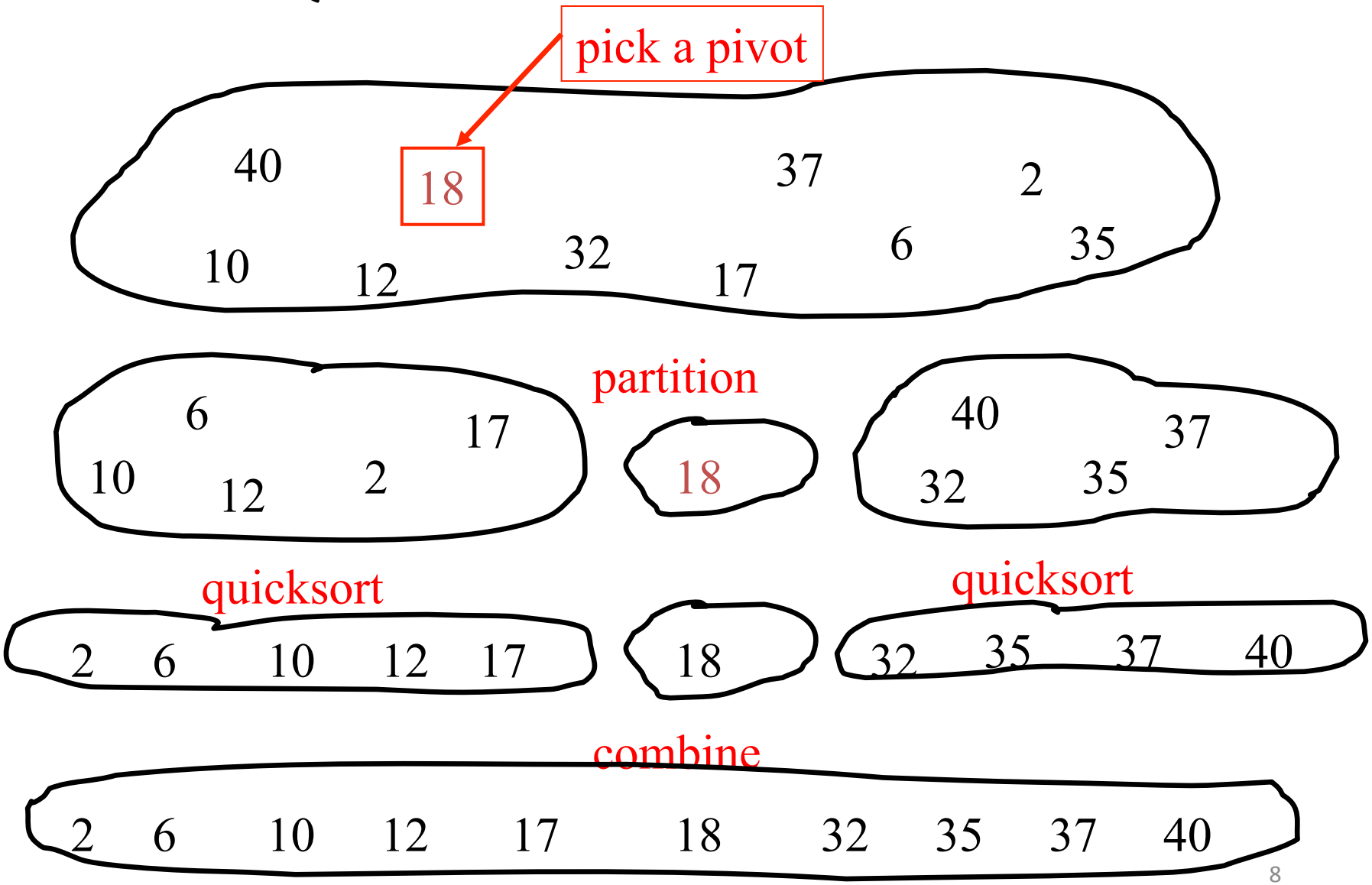
1. If $|S| = 0$ or $|S| = 1$, then return.

2. Pick an element v in S. Call v the pivot.

3. Partition $S - \{v\}$ into two disjoint groups:
   - $S_1 = \{x \in S - \{v\} \mid x \leq v\}$
   - $S_2 = \{x \in S - \{v\} \mid x \geq v\}$

4. Return { quicksort(S_1), v, quicksort(S_2) }

Quick sort illustrated



How to choose a pivot

- first element
  - bad if input is sorted or in reverse sorted order
  - bad if input is nearly sorted
  - variation: particular element (e.g. middle element)
- random element
  - even a malicious agent cannot arrange a bad input
- median of three elements
  - choose the median of the left, right, and center elements

Partitioning algorithm

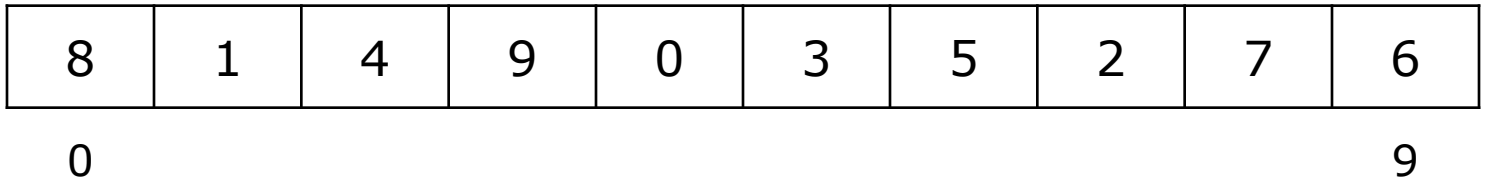
The basic idea:

1. Move the pivot to the rightmost position.
2. Starting from the left, find an element ≥ pivot. Call the position i.
3. Starting from the right, find an element ≤ pivot. Call the position j.
4. Swap S[i] and S[j].

8	1	4	9	0	3	5	2	7	6

0 9

Partitioning example



Quick sort code

```
public static void quickSort(int[] a) {
    quickSort(a, 0, a.length - 1);
}

private static void quickSort(int[] a, int min, int max) {
    if (min >= max) {  // base case; no need to sort
        return;
    }

    // choose pivot -- we'll use the first element (might be bad!)
    int pivot = a[min];
    swap(a, min, max);  // move pivot to end

    // partition the two sides of the array
    int middle = partition(a, min, max - 1, pivot);

    // restore the pivot to its proper location
    swap(a, middle, max);

    // recursively sort the left and right partitions
    quickSort(a, min, middle - 1);
    quickSort(a, middle + 1, max);
}
```

Quick sort code, cont'd.

```
// partitions a with elements < pivot on left and
// elements > pivot on right;
// returns index of element that should be swapped with pivot
private static int partition(int[] a, int i, int j, int pivot) {
    i--;  j++;  // kludge because the loops pre-increment
    while (true) {
        // move index markers i,j toward center
        // until we find a pair of mis-partitioned elements
        do { i++; } while (i < j && a[i] < pivot);
        do { j--; } while (i < j && a[j] > pivot);

        if (i >= j) {
            break;
        } else {
            swap(a, i, j);
        }
    }

    return i;
}
```

Quick sort code, cont'd.

```
// partitions a with elements < pivot on left and
// elements > pivot on right;
// returns index of element that should be swapped with pivot
private static int partition(int[] a, int i, int j, int pivot) {
    i--;  j++;  // kludge because the loops pre-increment
    while (true) {
        // move index markers i,j toward center
        // until we find a pair of mis-partitioned elements
        do { i++; } while (i < j && a[i] < pivot);
        do { j--; } while (i < j && a[j] > pivot);

        if (i >= j) {
            break;
        } else {
            swap(a, i, j);
        }
    }

    return i;
}
```

Quick sort code, cont'd.

```
// partitions a with elements < pivot on left and
// elements > pivot on right;
// returns index of element that should be swapped with pivot
private static int partition(int[] a, int i, int j, int pivot) {
    i--;  j++;  // kludge because the loops pre-increment
    while (true) {
        // move index markers i,j toward center
        // until we find a pair of mis-partitioned elements
        do { i++; } while (i < j && a[i] < pivot);
        do { j--; } while (i < j && a[j] > pivot);

        if (i < j) {
            swap(a, i, j);
        } else {
            break;
        }
    }

    return i;
}
```

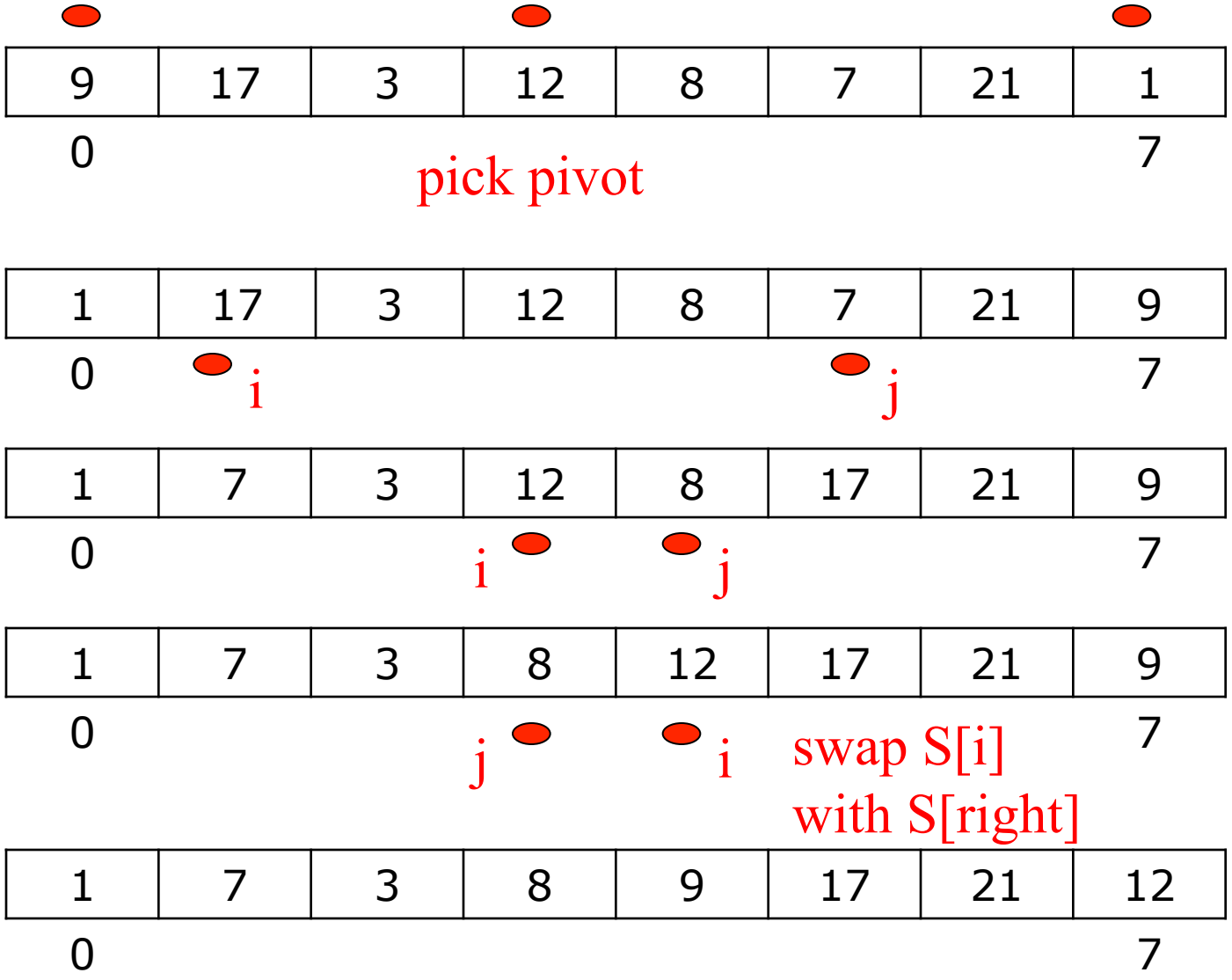
Quick sort code, cont'd.

```
// partitions a with elements < pivot on left and
// elements > pivot on right;
// returns index of element that should be swapped with pivot
private static int partition(int[] a, int i, int j, int pivot) {
    i--;  j++;  // kludge because the loops pre-increment
    while (true) {
        // move index markers i,j toward center
        // until we find a pair of mis-partitioned elements
        do { i++; } while (i < j && a[i] < pivot);
        do { j--; } while (i < j && a[j] > pivot);

        if (i < j) {
            swap(a, i, j);
        } else {
            break;
        }
    }

    return i;
}
```

"Median of three" pivot



Special cases

- What happens when the array contains many duplicate elements?
- What happens when the array is already sorted (or nearly sorted) to begin with?
- Small arrays
  - Quicksort is slower than insertion sort when is N is small (say, $N \leq 20$).
  - *Optimization*: Make $|A| \leq 20$ the base case and use insertion sort algorithm on arrays of that size or smaller.

Quick sort runtime

- Worst case: pivot is the smallest (or largest) element all the time (recurrence solution technique: telescoping)

  T(*n*) = T(*n*-1) + c*n*

  T(*n*-1) = T(*n*-2) + c(*n*-1)

  T(*n*-2) = T(*n*-3) + c(*n*-2)

  …

  T(2) = T(1) + 2c

  $$T(N) = T(1) + c \sum_{i=2}^{N} i = O(N^2)$$

- Best case: pivot is the median (recurrence solution technique: Master's Theorem)

  T(*n*) = 2 T(*n*/2) + c*n*

  T(*n*) = c*n* log *n* + *n* = O(*n* log *n*)

Quick sort runtime, cont'd.

- Assume each of the sizes for S_1 are equally likely. $0 \leq |S_1| \leq N-1$.

$$T(N) = \left(\frac{1}{N} \sum_{i=0}^{N-1} [T(i) + T(N - i - 1)] \right) + cN$$

$$= \left(\frac{2}{N} \sum_{i=0}^{N-1} T(i) \right) + cN$$

$$N\,T(N) = \left(2 \sum_{i=0}^{N-1} T(i) \right) + cN^2$$

$$(N-1)\,T(N-1) = 2 \sum_{i=0}^{N-2} T(i) + \quad c(N-1)^2$$

Quick sort runtime, cont'd.

$$N\,T(N) = (N+1)\,T(N-1) + 2cN$$

divide equation by N(N+1)

$$\frac{T(N)}{N+1} = \frac{T(N-1)}{N} + \frac{2c}{N+1}$$

$$\frac{T(N-1)}{N} = \frac{T(N-2)}{N-1} + \frac{2c}{N}$$

$$\frac{T(N-2)}{N-1} = \frac{T(N-3)}{N-2} + \frac{2c}{N-1}$$

$$\cdots$$

$$\frac{T(2)}{3} = \frac{T(1)}{2} + \frac{2c}{3}$$

$$\frac{T(N)}{N+1} = \frac{T(1)}{2} + 2c\sum_{i=3}^{N+1}\frac{1}{i}$$

$\approx \log_e (N+1) - 3/2$

$$T(N) = O(N \log N)$$

Quick sort runtime summary

- O(*n* log *n*) on average.
- O(n^2) worst case.

	comparisons
merge	O(n log n)
quick	average: O(n log n) worst: O(n^2)

Sorting practice problem

- Consider the following array of int values.

```
[22, 11, 34, -5,  3, 40,  9, 16,  6]
```

(f) Write the contents of the array after all the partitioning of quick sort has finished (before any recursive calls).

Assume that the median of three elements (first, middle, and last) is chosen as the pivot.

Sorting practice problem

- Consider the following array:

  `[ 7, 17, 22, -1,  9,  6, 11, 35, -3]`

- Each of the following is a view of a sort-in-progress on the elements. Which sort is which?
  - (If the algorithm is a multiple-loop algorithm, the array is shown after a few of these loops have completed. If the algorithm is recursive, the array is shown after the recursive calls have finished on each sub-part of the array.)
  - Assume that the quick sort algorithm chooses the ***first*** element as its pivot at each pass.

(a) `[-3, -1,  6, 17,  9, 22, 11, 35,  7]`

(b) `[-1,  7, 17, 22, -3,  6,  9, 11, 35]`

(c) `[ 9, 22, 17, -1, -3,  7,  6, 35, 11]`

(d) `[-1,  7,  6,  9, 11, -3, 17, 22, 35]`

(e) `[-3,  6, -1,  7,  9, 17, 11, 35, 22]`

(f) `[-1,  7, 17, 22,  9,  6, 11, 35, -3]`

Sorting practice problem

- For the following questions, indicate which of the six sorting algorithms will successfully sort the elements in the least amount of time.
  - The algorithm chosen should be the one that completes fastest, *without crashing*.
  - Assume that the quick sort algorithm chooses the ***first*** element as its pivot at each pass.
  - Assume stack overflow occurs on 5000+ stacked method calls.
  - (a) array size 2000, random order
  - (b) array size 500000, ascending order
  - (c) array size 100000, descending order
    - special constraint: no extra memory may be allocated! (O(1) storage)
  - (d) array size 1000000, random order
  - (e) array size 10000, ascending order
    - special constraint: no extra memory may be allocated! (O(1) storage)

Lower Bound for Comparison Sorting

- Theorem: Any algorithm that sorts using only comparisons between elements requires

  $\Omega(n \log n)$ comparisons.
  - Intuition
    - n! permutations that a sorting algorithm can output
    - each new comparison between any elements a and b cuts down the number of possible valid orderings by at most a factor of 2 (either all orderings where a > b or orderings where b > a)
    - to know which output to produce, the algorithm must make at least $\log_2(n!)$ comparisons before
    - $\log_2(n!) = \Omega(n \log n)$

O(n) Specialized Sorting

Bucket sort

- The bucket sort makes assumptions about the data being sorted
- Consequently, we can achieve better than $\Theta(n \log n)$ run times

Bucket Sort: Supporting Example

- Suppose we are sorting a large number of local phone numbers, for example, all residential phone numbers in the 206 area code region (over two million)

- Consider the following scheme:
  - create an array with $10\,000\,000$ bits (i.e. `BitSet`)
  - set each bit to 0 (indicating false)
  - for each phone number, set the bit indexed by the phone number to 1 (true)
  - once each phone number has been checked, walk through the array and for each bit which is 1, record that number

Bucket Sort: Supporting Example

- In this example, the number of phone numbers ($2\ 000\ 000$) is comparable to the size of the array ($10\ 000\ 000$)
- The run time of such an algorithm is $\mathbf{O}(n)$:
  - we make one pass through the data,
  - we make one pass through the array and extract the phone numbers which are true

Bucket Sort

- This approach uses very little memory and allows the entire structure to be kept in main memory at all times
- We will term each entry in the bit array a *bucket*
- We fill each bucket as appropriate

Example

- Consider sorting the following set of unique integers in the range 0, ..., 31:

20 1 31 8 29 28 11 14 6 16 15

27 10 4 23 7 19 18 0 26 12 22

- Create an bit array with 32 buckets
- This requires 4 bytes

Example

- For each number, set the bit of the corresponding bucket to 1
- Now, just traverse the list and record only those numbers for which the bit is 1 (true):

0 1 4 6 7 8 10 11 12 14 15

16 18 19 20 22 23 26 27 28 29 31

Bucket Sort

- How is this so fast?
- An algorithm which can sort arbitrary data must be
  $\Omega(n \log n)$
- In this case, we don't have arbitrary data: we have one further constraint, that the items being sorted fall within a certain range
- Using this assumption, we can reduce the run time

Bucket Sort

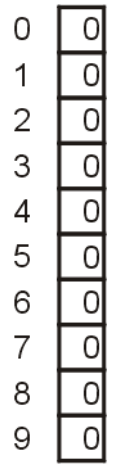
- Modification: what if there are repetitions in the data
- In this case, a bit array is insufficient
- Two options, each bucket is either:
  - a counter, or
  - a linked list
- The first is better if objects in the bin are the same

Example

- Sort the digits

0 3 2 8 5 3 7 5 3 2 8 2 3 5 1 3 2 8 5 3 4 9 2 3 5 1 0 9 3 5 2 3 5 4 2 1 3

- We start with an array of 10 counters, each initially set to zero:



Example

- Moving through the first 10 digits

0 3 2 8 5 3 7 5 3 2 8 2 3 5 1 3 2 8 5 3 4 9 2 3 5 1 0 9 3 5 2 3 5 4 2 1 3

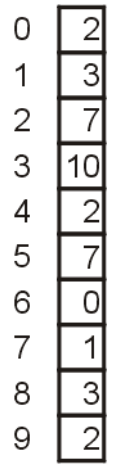
we increment the corresponding buckets

Example

- Moving through remaining digits

0 3 2 8 5 3 7 5 3 2 8 2 3 5 1 3 2 8 5 3 4 9 2 3 5 1 0 9 3 5 2 3 5 4 2 1 3

we continue incrementing the corresponding buckets



Example

- We now simply read off the number of each occurrence:

0 0 1 1 1 2 2 2 2 2 2 2 3 3 3 3 3 3 3 3 3 3 4 4 5 5 5 5 5 5 5 7 8 8 8 9 9

- For example:
  - there are seven 2s
  - there are two 4s

0	2
1	3
2	7
3	10
4	2
5	7
6	0
7	1
8	3
9	2

Run-time Summary

- The following table summarizes the run-times of bucket sort

Case	Run Time	Comments
Worst	$\Theta(n+m)$	No worst case
Average	$\Theta(n+m)$	
Best	$\Theta(n+m)$	No best case

External Sorting

Simple External Merge Sort

- Divide and conquer: divide the file into smaller, sorted subfiles (called runs) and merge runs

- Initialize:
  - Load chunk of data from file into RAM
  - Sort internally
  - Write sorted data (run) back to disk (in separate files)
- While we still have runs to sort:
  - Merge runs from previous pass into runs of twice the size (think merge() method from mergesort)
  - Repeat until you only have one run