Announcements

- H-006 due Friday
  ↳ look at previous feedback
    → Both on Gradescope + on D2L
      (one copy per group)

    → n+1st assignment: keep in mind
      demonstrate how you've grown

- Discussion Board!

Sorting:

- Bubble Sort $\Theta(n^2)$ ← not optimal
- Quicksort: In expectation $O(n\log n)$
              Worst case: $O(n^2)$
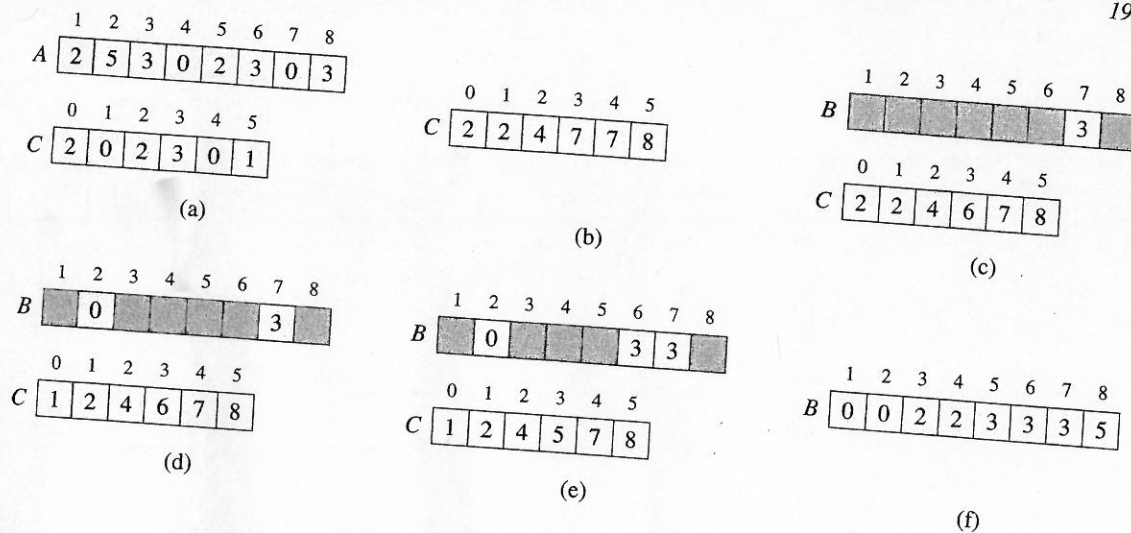
- Merge Sort $\Theta(n\log n)$
↑ all comparison-based!

A (1–8): 2 5 3 0 2 3 0 3
C (0–5): 2 0 2 3 0 1

(a)

C (0–5): 2 2 4 7 7 8

(b)

B (1–8): _ _ _ _ _ _ 3 _
C (0–5): 2 2 4 6 7 8

(c)

B (1–8): _ 0 _ _ _ _ 3 _
C (0–5): 1 2 4 6 7 8

(d)

B (1–8): _ 0 _ _ _ 3 3 _
C (0–5): 1 2 4 5 7 8

(e)

B (1–8): 0 0 2 2 3 3 3 5

(f)

Figure 8.2 The operation of COUNTING-SORT on an input array $A[1 . . 8]$, where each element of A is a nonnegative integer no larger than $k = 5$. (a) The array A and the auxiliary array C after line 5. (b) The array C after line 8. (c)–(e) The output array B and the auxiliary array C after one, two, and three iterations of the loop in lines 10–12, respectively. Only the lightly shaded elements of array B have been filled in. (f) The final sorted output array B.

COUNTING-SORT(A, B, k)

```
 1  let C[0 . . k] be a new array
 2  for i = 0 to k
 3      C[i] = 0
 4  for j = 1 to A.length
 5      C[A[j]] = C[A[j]] + 1
 6  // C[i] now contains the number of elements equal to i.
 7  for i = 1 to k
 8      C[i] = C[i] + C[i − 1]
 9  // C[i] now contains the number of elements less than or equal to i.
10  for j = A.length downto 1
11      B[C[A[j]]] = A[j]
12      C[A[j]] = C[A[j]] − 1
```

Figure 8.2 illustrates counting sort. After the **for** loop of lines 2–3 initializes the array C to all zeros, the **for** loop of lines 4–5 inspects each input element. If the value of an input element is i, we increment $C[i]$. Thus, after line 5, $C[i]$ holds the number of input elements equal to i for each integer $i = 0, 1, \ldots, k$. Lines 7–8 determine for each $i = 0, 1, \ldots, k$ how many input elements are less than or equal to i by keeping a running sum of the array C.

329	720	720	329
457	355	329	355
657	436	436	436
839	457	839	457
436	657	355	657
720	329	457	720
355	839	657	839

Figure 8.3 The operation of radix sort on a list of seven 3-digit numbers. The leftmost column is the input. The remaining columns show the list after successive sorts on increasingly significant digit positions. Shading indicates the digit position sorted on to produce each list from the previous one.

In a typical computer, which is a sequential random-access machine, we sometimes use radix sort to sort records of information that are keyed by multiple fields. For example, we might wish to sort dates by three keys: year, month, and day. We could run a sorting algorithm with a comparison function that, given two dates, compares years, and if there is a tie, compares months, and if another tie occurs, compares days. Alternatively, we could sort the information three times with a stable sort: first on day, next on month, and finally on year.

The code for radix sort is straightforward. The following procedure assumes that each element in the n-element array A has d digits, where digit 1 is the lowest-order digit and digit d is the highest-order digit.

RADIX-SORT(A, d)

1   **for** $i = 1$ **to** d
2   use a stable sort to sort array A on digit i

Lemma 8.3
Given n d-digit numbers in which each digit can take on up to k possible values, RADIX-SORT correctly sorts these numbers in $\Theta(d(n + k))$ time if the stable sort it uses takes $\Theta(n + k)$ time.

Proof   The correctness of radix sort follows by induction on the column being sorted (see Exercise 8.3-3). The analysis of the running time depends on the stable sort used as the intermediate sorting algorithm. When each digit is in the range 0 to $k-1$ (so that it can take on k possible values), and k is not too large, counting sort is the obvious choice. Each pass over n d-digit numbers then takes time $\Theta(n + k)$. There are d passes, and so the total time for radix sort is $\Theta(d(n + k))$. ∎

When d is constant and $k = O(n)$, we can make radix sort run in linear time. More generally, we have some flexibility in how to break each key into digits.
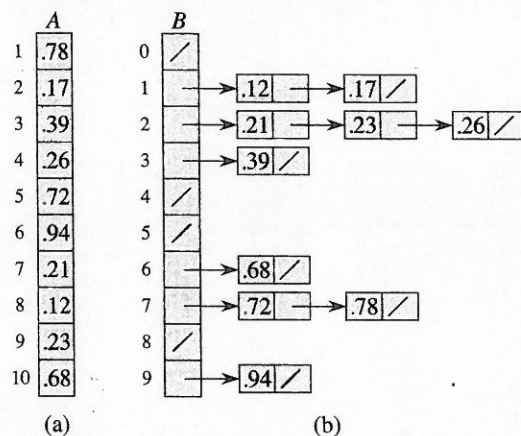
Figure 8.4 The operation of BUCKET-SORT for $n = 10$. **(a)** The input array $A[1 .. 10]$. **(b)** The array $B[0 .. 9]$ of sorted lists (buckets) after line 8 of the algorithm. Bucket i holds values in the half-open interval $[i/10, (i + 1)/10)$. The sorted output consists of a concatenation in order of the lists $B[0], B[1], \ldots, B[9]$.

BUCKET-SORT(A)

```
1   n = A.length
2   let B[0..n − 1] be a new array
3   for i = 0 to n − 1
4       make B[i] an empty list
5   for i = 1 to n
6       insert A[i] into list B[⌊nA[i]⌋]
7   for i = 0 to n − 1
8       sort list B[i] with insertion sort
9   concatenate the lists B[0], B[1], . . . , B[n − 1] together in order
```

Figure 8.4 shows the operation of bucket sort on an input array of 10 numbers.

To see that this algorithm works, consider two elements $A[i]$ and $A[j]$. Assume without loss of generality that $A[i] \leq A[j]$. Since $\lfloor nA[i] \rfloor \leq \lfloor nA[j] \rfloor$, either element $A[i]$ goes into the same bucket as $A[j]$ or it goes into a bucket with a lower index. If $A[i]$ and $A[j]$ go into the same bucket, then the **for** loop of lines 7–8 puts them into the proper order. If $A[i]$ and $A[j]$ go into different buckets, then line 9 puts them into the proper order. Therefore, bucket sort works correctly.

To analyze the running time, observe that all lines except line 8 take $O(n)$ time in the worst case. We need to analyze the total time taken by the n calls to insertion sort in line 8.