Comparison Sorts

- Bubble Sort, Insertion Sort, Merge Sort, and Heap Sort are all **Comparison Sorts** (order determined by comparing elements)

- Counting sort, Radix sort, and Bucket sort are algorithms that work by not **directly** comparing the elements

- Benefit: Much better $\Omega(\cdot)$ bound
Comparison Sort: $A = <6,8,5>$

Note: Leaves to produce sort tree for array of size $n$?

Comparison Sort: $\Omega$ bound

- Theorem 8.1: Any Comparison Sort makes $\Omega(n \lg n)$ comparisons.
- Pf:

- Corollary 8.2: Heapsort and MergeSort are as good as you can do for a Comparison Sort

\[ n! \sim \sqrt{2\pi n} \left(\frac{n}{e}\right)^n \]

Stirling’s Formula
Counting Sort

- Depends on a key assumption:
  - numbers to be sorted are integers in \{0, 1, ..., k\}
- **Input:** \(A[1..n]\)
- **Output:** \(B[1..n]\), sorted. \(B\) is assumed to be already allocated and is given as a parameter
- **Auxiliary storage:** \(C[0..k]\)

- Idea: For each element \(A[i]\), count number of elements less than it and use to put in correct position in sorted array \(B\)

---

**Example**

\[2_1, 5_1, 3_1, 0_1, 2_2, 3_2, 0_2, 3_3\]

\[\begin{array}{cccccccc}
A & 2 & 5 & 3 & 0 & 2 & 3 & 0 & 3 \\
 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 \\
C & 2 & 0 & 2 & 3 & 0 & 1 \\
\end{array}\]

Count each value \(A[i]\), store in array at position \(A[i]\)

\[\begin{array}{cccccccc}
B & & & & & & & & \\
 & 3 \\
 & 0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 \\
C & 2 & 2 & 4 & 7 & 7 & 8 \\
\end{array}\]

Count input values that are less than or equal to value

Using counts, put each value in proper position in array, decrement

Next?...then Next?
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 = 0 to A.length
6.     // so C[i] contains # elements = i
7. for i = 1 to k
8.     C[i] = C[i] + C[i-1]
9.     // so C[i] contains # elements <= i
10. for j = A.length to 1

Analysis

- Is counting sort stable?
  - Numbers with same value appear in same order in output array as they did in input array
- Analysis:

- How big of k is practical?
Your Turn

- A: <6, 0, 2, 0, 1, 3, 4, 6, 1, 3, 2>

Radix Sort

- How IBM made its money. Punch card readers for census tabulation in early 1900’s. Card sorters, worked on one column at a time. It’s the algorithm for using the machine that extends the technique to multi-column sorting. The human operator was part of the algorithm!

- We’re going to sort numbers with d-digits, but Radix Sort will also sort words, dates by year, month, day, etc.

RADIX-SORT(A, d)

1  for i = 1 to d
2  use any stable sort to sort the Array A on digit i (least significant digit (LSD) first)

Analysis:

Example

<table>
<thead>
<tr>
<th>326</th>
<th>690</th>
</tr>
</thead>
<tbody>
<tr>
<td>453</td>
<td>751</td>
</tr>
<tr>
<td>608</td>
<td>453</td>
</tr>
<tr>
<td>835</td>
<td>704</td>
</tr>
<tr>
<td>751</td>
<td></td>
</tr>
<tr>
<td>435</td>
<td></td>
</tr>
<tr>
<td>704</td>
<td></td>
</tr>
<tr>
<td>690</td>
<td></td>
</tr>
</tbody>
</table>
Bucket Sort

- Assumption: input is generated by a random process that distributes elements uniformly over [0,1)

- Idea:

- Analysis: $\theta(n)$ by random variable argument

---

Bucket Sort

- Input: A[1..n], where for all i

- Auxiliary array: B[0..n-1] of linked lists, each list initially empty.
BUCKET-SORT(A)

1. \( n = A.length \)
2. let \( B = [0...n-1] \) be new array
3. for \( i = 0 \) to \( n-1 \)
4. make \( B[i] \) empty list
5. for \( i = 1 \) to \( n \)
6. insert \( A[i] \) into list \( B[\lfloor nA[i]\rfloor] \)
7. for \( i = 0 \) to \( n-1 \)
8. sort list \( B[i] \) using insertion sort
9. Concatenate lists \( B[0], B[1], ..., B[n-1] \) in order

Example

- \( A: <.78, .17, .39, .26, .72, .94, .21, .12, .23, .68> \)

(a) 

(b)