Algorithms & Data Structures
Part 2

The Sorting Problem
This Course

• **Sorting**
  – A necessary tool for managing and navigating a variety of datasets.
  – A great context in which to learn computer science
    • Analyze Complexity
    • Prove Correctness

• **Data Structures**
  – Explore variety of structures beyond arrays and linked lists.
  – How you store data affects how easily you can manage and navigate it.

• **Optimization Problems**
  – Learn sophisticated methods that are alternates to exponential brute force approach.

• **Graph Problems**
  – Applicable to wide range of problems (many seemingly unrelated problems map to a graph problem).
  – Another great context in which to learn computer science.
### Sorting and Complexity

#### Comparison Sort
(No assumptions about data)
- Insertion Sort (Ch. 2)
- Merge Sort (Ch. 2)
- Heapsort (Ch. 6)
- Quicksort (Ch. 7)

#### Non-comparison Sort
(Special assumptions about data)
- Counting Sort (Ch. 8)
- Bucket Sort (Ch. 8)
- Radix Sort (Ch. 8)

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Description</th>
<th>Worst</th>
<th>Average</th>
</tr>
</thead>
<tbody>
<tr>
<td>Insertion Sort</td>
<td>Iteratively insert element into sorted list.</td>
<td>Θ(n^2)</td>
<td>Θ(n^2)</td>
</tr>
<tr>
<td>Merge Sort</td>
<td>Recursively divide, then merge 2 sorted lists.</td>
<td>Θ(nlgn)</td>
<td>Θ(nlgn)</td>
</tr>
<tr>
<td>Heapsort</td>
<td>Continually extract max off heap, heapify</td>
<td>O(nlgn)</td>
<td>--</td>
</tr>
<tr>
<td>Quicksort</td>
<td>Use “middle” to divide, then sort each</td>
<td>Θ(n^2)</td>
<td>Θ(nlgn)</td>
</tr>
<tr>
<td>Counting Sort</td>
<td>Record frequency of each #, place accordingly</td>
<td>Θ(k + n)</td>
<td>Θ(k + n)</td>
</tr>
<tr>
<td>Radix Sort</td>
<td>Sort on each “digit” using stable sort</td>
<td>Θ(d(n + k))</td>
<td>Θ(d(n + k))</td>
</tr>
</tbody>
</table>
Heapsort

Binary Heap

– Visualized as nearly complete binary tree.
– Stored in an array.
– Maintains max-heap property.

\[ A[\text{PARENT}(i)] \geq A[i] \text{ for all nodes not the root} \]

Properties of Max-Heap

– Largest element at root.
– Every node is root of subtree and largest element of subtree.
MAX-HEAPIFY
Correctly Place 1 Specific Element

6 18 12 15 10 11 5 7 9 3

Draw array as binary tree, then Heapify(A,1).

MAX-HEAPIFY(A, i) assumes LEFT(i) and RIGHT(i) are max-heaps

MAX-HEAPIFY(A, i)
1 \(l = \text{LEFT}[i]\)
2 \(r = \text{RIGHT}[i]\)
3 \(\text{if } l \leq A.\text{heap-size} \text{ and } A[l] > A[i]\)
4 \(\text{largest} = l\)
5 \(\text{else} \quad \text{largest} = i\)
6 \(\text{if } r \leq A.\text{heap-size} \text{ and } A[r] > A[\text{largest}]\)
7 \(\text{largest} = r\)
8 \(\text{if } \text{largest} \neq i\)
9 \(\text{exchange } A[i] \text{ with } A[\text{largest}]\)
10 \(\text{MAX-HEAPIFY}(A, \text{largest})\)

DESCRIPTION:
Given element A[i], “sink” A[i] towards the leaves to its proper position.

PROCESS:
2. If parent and child swapped (ie A[i] sunk down a level), heapify child location.
What is the running time?

Recursive Call => Need Recurrence

MAX-HEAPIFY(A, i)

1 \( i = \text{LEFT}[i] \)
2 \( r = \text{RIGHT}[i] \)
3 if \( i \leq \text{A.heap-size} \) and \( A[i] > A[i] \)
4 \( \text{largest} = i \)
5 else \( \text{largest} = i \)
6 if \( r \leq \text{A.heap-size} \) and \( A[r] > A[\text{largest}] \)
7 \( \text{largest} = r \)
8 if \( \text{largest} \neq i \)
9 exchange \( A[i] \) with \( A[\text{largest}] \)
10 MAX-HEAPIFY(A, \text{largest})
### Master Theorem

\[ T(n) = a^*T(n/b) + f(n) \]

<table>
<thead>
<tr>
<th>Bound on f(n)</th>
<th>Growth of T(n)</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>( f(n) = O(n^{\log_b a - \varepsilon}) ) for some ( \varepsilon &gt; 0 )</td>
<td>( T(n) = \Theta(n^{\log_b a}) )</td>
<td>leaves dominate</td>
</tr>
<tr>
<td>( f(n) = \Theta(n^{\log_b a}) )</td>
<td>( T(n) = \Theta(n^{\log_b a \cdot \log n}) )</td>
<td>leaves and nodes same</td>
</tr>
<tr>
<td>( f(n) = \Omega(n^{\log_b a + \varepsilon}) ) for some ( \varepsilon &gt; 0 ) AND IF ( af(n/b) \leq cf(n) ) for some ( c &lt; 1 )</td>
<td>( T(n) = \Theta(f(n)) )</td>
<td>nodes dominate</td>
</tr>
</tbody>
</table>
BUILD-MAX-HEAP
Correctly Place ALL Elements

Correctly place root of each subtree using MAX-HEAPIFY, starting at the “lowest” subtree.

“Lowest” subtree is rooted at parent of last element.

Move “right to left” across each level, considering the subtree rooted at each of those nodes.

Each child of each newly evaluated node is the root of a Max Heap. Once root is placed, the subtree is a Max Heap.
Start at “lowest” subtree

Consider nodes (subtrees) right to left.

Each child is root of Max Heap. Root of subtree is only node misplaced.

**BUILD-MAX-HEAP(A)**

1. A.heap-size = A.length
2. for i = ⌊A.length/2⌋ down to 1
3. MAX-HEAPIFY(A, i)
Proof of Correctness for BUILD-MAX-HEAP

INPUT: A sequence of n numbers \{ a_1, a_2, a_3, \ldots, a_n \}

OUTPUT: An array (max-heap) containing \{ a_1, a_2, a_3, \ldots, a_n \}
such that for all elements (nodes) A[i]:
A[\text{Parent}(i)] \geq A[i]

Loop Invariant
“At the start of each iteration of the for loop,
each node i+1, i+2, \ldots, n is the root of a max-heap.”

\begin{verbatim}
BUILD-MAX-HEAP(A)
1   A.heap-size = A.length
2   for i = \lfloor A.length/2 \rfloor \text{ downto } 1
3       MAX-HEAPIFY(A, i)
\end{verbatim}

\(O(n) \text{ not } O(n \log n)\)

Note that we still have not produced a sorted list!
Proof of Correctness for BUILD-MAX-HEAP

\[
\sum_{h=0}^{\lceil \lg n \rceil} \left\lfloor \frac{n}{2^h+1} \right\rfloor O(h) = O\left(n \sum_{h=0}^{\lceil \lg n \rceil} \frac{h}{2^h}\right).
\]

We evaluate the last summation by substituting \( x = 1/2 \) in the formula (A.8), yielding

\[
\sum_{h=0}^{\infty} \frac{h}{2^h} = \frac{1/2}{(1 - 1/2)^2} = 2.
\]

Thus, we can bound the running time of BUILD-MAX-HEAP as

\[
O\left(n \sum_{h=0}^{\lceil \lg n \rceil} \frac{h}{2^h}\right) = O\left(n \sum_{h=0}^{\infty} \frac{h}{2^h}\right) = O(n).
\]

Note that we still have not produced a sorted list!
HEAPSORT

Build a heap from a randomly ordered array of numbers, then sort.

Essentially build the list in reverse order, by placing the root (max) in the list.
Then finding the new root with HEAPIFY.

This sorts IN PLACE

\[ O(n \log n) \]
Heaps Used to Maintain a Dynamic Set (e.g. Scheduling)

MAXIMUM(S) : S[1]

EXTRACT-MAX(S) :
   swap S.1 and S.length.
   S.length = S.length-1
   Heapify(S.1)

INCREASE-KEY(S,x,k)
   Set S[x] = key (which is greater than it was)
   Float up key to appropriate place by comparing with parent

INSERT(S,x)
   S.length = S.length+1.
   S.length = -infinity.
   Increase-Key( S, S.length, k )
HEAP-INCREASE-KEY(A, i, key)
1 if key < A[i]
2   error "new key is smaller than current key"
3   A[i] = key
4 while i > 1 and A[PARENT(i)] < A[i]
5   exchange A[i] with A[PARENT(i)]
6   i = PARENT(i)
Various Sorting Algorithms

- **Comparison Sort (No assumptions about data)**
  - Insertion Sort (Ch. 2)
  - Merge Sort (Ch. 2)
  - Heapsort (Ch. 6)
  - **Quicksort (Ch. 7)**

- **Non-comparison Sort (Special assumptions about data)**
  - Counting Sort (Ch. 8)
  - Bucket Sort (Ch. 8)
  - Radix Sort (Ch. 8)
Sort papers by last name:

Compare each paper to middle letter ‘M’
Pile papers A-M on the left (NOT ordered)
Pile papers N-Z on the right.

Pick up A-M (compare to ‘G’):
Pile papers A-G on the left.
Pile papers F-M on the right.

Pick up A-B (compare to ‘A’)
Pile papers A on the left.
Pile papers B on the right.

Stack pile A on top of pile B.
Stack sorted pile AB on top of sorted pile CD.

Stack sorted pile A-M on top of sorted pile N-Z
QUICKSORT \((A, p, r)\)

1. if \(p < r\)

2. \(q = \text{PARTITION} (A, p, r)\)

3. \(\text{QUICKSORT} (A, p, q - 1)\)

4. \(\text{QUICKSORT} (A, q + 1, r)\)

PARTITION \((A, p, r)\)

1. \(x = A[r]\)

2. \(i = p - 1\)

3. for \(j = p\) to \(r - 1\)

4. if \(A[j] \leq x\)

5. \(i = i + 1\)

6. exchange \(A[i]\) with \(A[j]\)

7. exchange \(A[i + 1]\) with \(A[r]\)

8. return \(i + 1\)

- Arbitrarily choose last element as “middle value”
- \(i\) holds place for one left of “middle index”
- Rearrange array so
  - pile 1 = \(A[p].A[\text{“middle”-1}]\)
  - pile 2 = \(A[\text{“middle”+1}]..A[r]\)
- Piles not sorted
- Properly place “middle”
QUICKSORT( A, 1, 8 )
4 = PARTITION( A, 1, 8 )
QUICKSORT( A, 1, 3 )
QUICKSORT( A, 5, 8 )

depicts PARTITION( A, 1, 8 ) which returns INDEX #4.

"Middle" value 4 is now properly placed at index 4.
Value = Index because the list is {1..8}.
QUICKSORT ($A$, $p$, $r$)
1 if $p < r$
2 $q =$ PARTITION ($A$, $p$, $r$)
3 QUICKSORT ($A$, $p$, $q - 1$)
4 QUICKSORT ($A$, $q + 1$, $r$)

PARTITION ($A$, $p$, $r$)
1 $x = A[r]$
2 $i = p - 1$
3 for $j = p$ to $r - 1$
4 if $A[j] \leq x$
5 $i = i + 1$
6 exchange $A[i]$ with $A[j]$
7 exchange $A[i + 1]$ with $A[r]$
8 return $i + 1$

At the beginning of each iteration of the loop of lines 3–6, for any array index $k$,

1. If $p \leq k \leq i$, then $A[k] \leq x$.
2. If $i + 1 \leq k \leq j - 1$, then $A[k] > x$.
3. If $k = r$, then $A[k] = x$. 

Loop Invariant
QUICKSORT(A, 1, 8)
4 = PARTITION(A, 1, 8)
QUICKSORT(A, 1, 3)
QUICKSORT(A, 5, 8)

BEST: T(n) =
WORST: T(n) =
AVERAGE: T(n) = ????
• **Average Running Time**
  
  – All $n!$ permutations of $A$ are equally likely.
  – On average, the splits will be a mix of good and bad.
  – Intuitively, characterize as worst, best, worst, best, ...
  – $T(n) = T(0) + T(\frac{n-1}{2}) + T(\frac{n-1}{2} - 1)$
    
    \[ \Rightarrow 2*T(\frac{n-1}{2}) + cn; \]
    
    \[ \Rightarrow O(n\log n) \]

• **Expected Running Time of RANDOM Quicksort**
  
  – Randomly select “middle”
  – All elements equally likely to be selected as “middle” or “pivot”
  – Expected running time depends on likelihood of 2 elements being compared.
Expected Values

• What is expected value of random variable \( X \) ?
  
  \[
  E[X] = \sum (x \cdot \text{Pr}\{X = x\}).
  \]
  
  – \( X \) in range \( \{1 .. 5\} \).
  
  – \( \text{Pr}\{X = x\} = 1/5 \), for all \( x \) \hspace{1cm} \text{(uniform probability distribution)}
  
  – \( E[X] = 1 \cdot 1/5 + 2 \cdot 1/5 + 3 \cdot 1/5 + 4 \cdot 1/5 = 3 \) \hspace{1cm} \text{(i.e. the mean)}

• Indicator Random Variable
  
  – \( I\{A\} = \{1: \text{if } A \text{ occurs. } 0: \text{if } A \text{ does not occur.}\} \)
  
  – \( X_A = I\{A\} \cdot E(X_A) = E[I\{A\}] \).
  
  – If there are a series of events, we can calculate the expected number of those events occurring ...
    
    \[
    X = \sum X_i.
    \]
    
    – \( E[X] = E[\sum X_i] = \sum E[X_i] \)
    
    \[
    = \sum 1 \cdot \text{Pr}\{X_i \text{ occurs }\} + 0 \cdot \text{Pr}\{X_i \text{ doesn’t occur}\}
    \]
    
    – \( E[X] = \sum \text{Pr}\{X_i \text{ occurs }\} \)
## Sorting and Complexity

### Comparison Sort
(No assumptions about data)
- Insertion Sort (Ch. 2)
- Merge Sort (Ch. 2)
- Heapsort (Ch. 6)
- Quicksort (Ch. 7)

### Non-comparison Sort
(Special assumptions about data)
- Counting Sort (Ch. 8)
- Bucket Sort (Ch. 8)
- Radix Sort (Ch. 8)

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Description</th>
<th>Worst</th>
<th>Average</th>
</tr>
</thead>
<tbody>
<tr>
<td>Insertion Sort</td>
<td>Iteratively insert element into sorted list.</td>
<td>$\Theta(n^2)$</td>
<td>$\Theta(n^2)$</td>
</tr>
<tr>
<td>Merge Sort</td>
<td>Recursively divide, then merge 2 sorted lists</td>
<td>$\Theta(n \log n)$</td>
<td>$\Theta(n \log n)$</td>
</tr>
<tr>
<td>Heapsort</td>
<td>Continually extract max off heap, heapify</td>
<td>$O(n \log n)$</td>
<td>--</td>
</tr>
<tr>
<td>Quicksort</td>
<td>Use “middle” to divide, then sort each</td>
<td>$\Theta(n^2)$</td>
<td>$\Theta(n \log n)$</td>
</tr>
<tr>
<td>Counting Sort</td>
<td>Record frequency of each #, place accordingly</td>
<td>$\Theta(k + n)$</td>
<td>$\Theta(k + n)$</td>
</tr>
<tr>
<td>Radix Sort</td>
<td>Sort on each “digit” using stable sort</td>
<td>$\Theta(d(n + k))$</td>
<td>$\Theta(d(n + k))$</td>
</tr>
<tr>
<td>Bucket Sort</td>
<td>Divide range into buckets, place accordingly</td>
<td>$\Theta(n^2)$</td>
<td>$\Theta(n)$</td>
</tr>
</tbody>
</table>
Limits of Comparison Sort

Decision Tree Model: Models all possible comparisons of insertion sort.


- All possible permutations appear in a leaf.


Length of path from root to leaf is number of comparisons (i.e. indication of running time.)

Lower Bound of All Comparison Sorts = MIN Height of All Decision Trees

All Comparison Sorts = Ω(nlgn)

in other words, if you sort by comparison, you can’t be more efficient than cnlgn.
Counting Sort

What You Need to Know About Your Data:
If data in range \{0 .. k\} AND k = O(n)
then counting sort Θ(n)

COUNTING-SORT(A, B, k)
1 let C[0 .. k] be a new array
2 for i = 0 to k
3 \quad C[i] = 0
4 for j = 1 to A.length
5 \quad 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 \quad 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 \quad B[C[A[j]]] = A[j]
12 \quad C[A[j]] = C[A[j]] - 1

k: max value
A[n]: original array
B[n]: sorted array
C[k]: counting array

Count #’s.
Accumulate Counts.
Place #’s in B.
Counting Sort Properties

```
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
12    C[A[j]] = C[A[j]] - 1
```

\[ f(n) = c_1 k + c_2 n + c_3 k + c_4 n \]
\[ = c_5 k + c_6 n \]
\[ = \Theta(n) \] if \( k = O(n) \)

**Counting Sort**
\[ f(n) = c*1,000,000,000 + d*10,000 \]

**Comparison Sort**
\[ f(n) = c*10,000*\lg 10,000 = c*140,000 \]
Running Time of Radix Sort

n: # of entries
k: max key value of all n’s
b: bits required to represent k
r: “word” size where r ≤ b, i.e. groupings of bits of YOUR choosing.

Lemma 8.4
For n b-bit keys and any r ≤ b,

\[ \text{RADIX-SORT} = \Theta((b/r)(n+2^r)) \]

This proof helps guide your choice for r ..

- if \( b < \text{floor}(\ lg \ n) \),
  - choose \( r = b \) \( \Rightarrow \Theta(n) \)
- if \( b \geq \text{floor}(\ lg \ n) \),
  - choose \( r = \text{floor}(\ lg \ n) \) \( \Rightarrow \Theta(bn / lg n) \)

**Counting Sort**
\[
\begin{align*}
f(n) &= c_1k + c_2n + c_3k + c_4n \\
&= c_5k + c_6n
\end{align*}
\]

**Radix Sort**
\[
\begin{align*}
f(n) &= c_7d \cdot (c_5k + c_6n) \\
&= \Theta(n) \text{ if } k = O(n)
\end{align*}
\]
<table>
<thead>
<tr>
<th>ss#, n=.5B Desc</th>
<th>b: bits</th>
<th>r: word</th>
<th>d: digits</th>
<th>k: max key</th>
<th>f(n)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Counting</td>
<td></td>
<td></td>
<td>9</td>
<td>1 billion</td>
<td>O( .5B + 1B )</td>
</tr>
<tr>
<td>Radix – Deci</td>
<td>--</td>
<td>1</td>
<td>9</td>
<td>10</td>
<td>O( 9(.5B + 10))</td>
</tr>
<tr>
<td>Radix – Binary</td>
<td>30</td>
<td>1</td>
<td>30</td>
<td>2</td>
<td>O( 30(.5B + 2))</td>
</tr>
<tr>
<td>Radix</td>
<td>30</td>
<td>30</td>
<td>1</td>
<td>1 billion</td>
<td>O( 1(.5B + 1B))</td>
</tr>
<tr>
<td>Radix</td>
<td>30</td>
<td>lg n = 28,(2)</td>
<td>2</td>
<td>268M, 4</td>
<td>O( (.5B + 268M) + (.5B + 4))</td>
</tr>
<tr>
<td>Comparison</td>
<td>--</td>
<td></td>
<td></td>
<td></td>
<td>O( .5B(lg .5B) )</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>5-digit ID, n=1M Desc</th>
<th>b: bits</th>
<th>r: word</th>
<th>d: digits</th>
<th>k: max key</th>
<th>f(n)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Counting</td>
<td></td>
<td></td>
<td>5</td>
<td>100,000</td>
<td>O( 1M + 100000)</td>
</tr>
<tr>
<td>Radix – Deci</td>
<td>--</td>
<td>1</td>
<td>5</td>
<td>10</td>
<td>O( 5( 1M + 10 ) )</td>
</tr>
<tr>
<td>Radix – Binary</td>
<td>17</td>
<td>1</td>
<td>17</td>
<td>2</td>
<td>O( 17( 1M + 2 ) )</td>
</tr>
<tr>
<td>Radix</td>
<td>17</td>
<td>17</td>
<td>1</td>
<td>100,000</td>
<td>O(1(1M+100000))</td>
</tr>
<tr>
<td>Radix</td>
<td>17</td>
<td>9</td>
<td>2</td>
<td>512</td>
<td>O(2(1M + 512))</td>
</tr>
<tr>
<td>Comparison</td>
<td>--</td>
<td></td>
<td></td>
<td></td>
<td>O( 1M(lg 1M) )</td>
</tr>
</tbody>
</table>
### Comparing Algorithms

<table>
<thead>
<tr>
<th>ss#, n=1500 Desc</th>
<th>b: bits</th>
<th>r: word</th>
<th>d: digits</th>
<th>k: max key</th>
<th>f(n)</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Counting</td>
<td></td>
<td></td>
<td>9</td>
<td>1 billion</td>
<td>$O(1500 + 1B)$</td>
<td>$\approx 1B$</td>
</tr>
<tr>
<td>Radix – Deci</td>
<td>--</td>
<td>1</td>
<td>9</td>
<td>10</td>
<td>$O(9(1500 + 10))$</td>
<td>$\approx 13,590$</td>
</tr>
<tr>
<td>Radix – Binary</td>
<td>30</td>
<td>1</td>
<td>30</td>
<td>2</td>
<td>$O(30(1500 + 2))$</td>
<td>$\approx 45,060$</td>
</tr>
<tr>
<td>Radix</td>
<td>30</td>
<td>30</td>
<td>1</td>
<td>1 billion</td>
<td>$O(1(1500 + 1B))$</td>
<td>$\approx 1B$</td>
</tr>
<tr>
<td>Radix</td>
<td>30</td>
<td>$\lg n = 10$</td>
<td>3</td>
<td>1024</td>
<td>$O(3(1500 + 1024))$</td>
<td>$\approx 7,572$</td>
</tr>
<tr>
<td>Comparison</td>
<td>--</td>
<td></td>
<td></td>
<td></td>
<td>$O(1500\lg 1500)$</td>
<td>$\approx 15,826$</td>
</tr>
</tbody>
</table>

#### Lemma 8.4
For n b-bit keys and any $r \leq b$,
$$\text{RADIX-SORT} = \Theta((b/r)(n+2r))$$

This proof helps guide your choice for $r$ ..
- if $b < \lfloor \lg n \rfloor$, choose $r = b \Rightarrow \Theta(n)$
- if $b \geq \lfloor \lg n \rfloor$, choose $r = \lfloor \lg n \rfloor \Rightarrow \Theta(bn / \lg n)$
Bucket Sort

What You Need to Know About Your Data:
If data is uniformly distributed in range \([0 .. 1)\)
then counting sort \(\Theta(n)\)

TA Sort: Sort Papers by Last Name

Make a pile for each letter of the alphabet.
Put paper in appropriate pile,
Sort each pile using insertion sort.
Put piles together, in order.

```
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[\(\lfloor nA[i] \rfloor\)].
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
```

“Proof” of Correctness
Running Time Bucket Sort

What You Need to Know About Your Data:
If data is uniformly distributed in range [ 0 .. 1 ) then counting sort $\Theta(n)$

```
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
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
```

$T(n) = \Theta(n) + \sum_{i=0}^{n-1} O(n_i^2)$

$E[T(n)] = E[ \Theta(n) + \sum_{i=0}^{n-1} O(n_i^2) ]$

$E[T(n)] = \Theta(n) + \sum_{i=0}^{n-1} E[ O(n_i^2) ]$

$E[T(n)] = \Theta(n) + \sum_{i=0}^{n-1} O( E[n_i^2] )$
Observations on Bucket Sort

• The implementation shown in the book uses pointers. What is the advantage of pointers in this situation?

• How would you change the algorithm if there were 100 integers in the range [0,100) (with uniform distribution)? How is this different from counting sort or radix sort?

• Consider the scenario with 1,000,000 numbers in the range [0,1). Would this have an advantage over other algorithms?

• What if there were 100 buckets but a 1,000 numbers. What would you expect as the sort time (intuitively)?

![Diagram of Bucket Sort](image)

```
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[\lfloor nA[i] \rfloor].
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
```
Comparing Algorithms

• Know your data
  – Distribution
  – Maximum value
  – Size of the input

• Know your algorithm
  – “(hidden) Constant factors differ ... each pass over the data of radix sort may take significantly longer (than a pass in quicksort).”
  – Choosing the right “r” to improve results.
  – Counting sort does NOT sort in place.
  – Radix depends on use of a stable, linear sorting algorithm.

• Know your machine/implementation
  – “quicksort often uses hardware caches more effectively than radix.”
  – What about multicore systems or parallel processing?
## Sorting and Complexity

<table>
<thead>
<tr>
<th>Comparison Sort</th>
<th>Non-comparison Sort</th>
</tr>
</thead>
<tbody>
<tr>
<td>(No assumptions about data)</td>
<td>(Special assumptions about data)</td>
</tr>
<tr>
<td>– Insertion Sort (Ch. 2)</td>
<td>– Counting Sort (Ch. 8)</td>
</tr>
<tr>
<td>– Merge Sort (Ch. 2)</td>
<td>– Bucket Sort (Ch. 8)</td>
</tr>
<tr>
<td>– Heapsort (Ch. 6)</td>
<td>– Radix Sort (Ch. 8)</td>
</tr>
<tr>
<td>– Quicksort (Ch. 7)</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Description</th>
<th>Worst</th>
<th>Average</th>
</tr>
</thead>
<tbody>
<tr>
<td>Insertion Sort</td>
<td>Iteratively insert element into sorted list.</td>
<td>$\Theta(n^2)$</td>
<td>$\Theta(n^2)$</td>
</tr>
<tr>
<td>Merge Sort</td>
<td>Recursively divide, then merge 2 sorted lists</td>
<td>$\Theta(n \log n)$</td>
<td>$\Theta(n \log n)$</td>
</tr>
<tr>
<td>Heapsort</td>
<td>Continually extract max off heap, heapify</td>
<td>$O(n \log n)$</td>
<td>--</td>
</tr>
<tr>
<td>Quicksort</td>
<td>Use “middle” to divide, then sort each</td>
<td>$\Theta(n^2)$</td>
<td>$\Theta(n \log n)$</td>
</tr>
</tbody>
</table>

| Counting Sort      | Record frequency of each #, place accordingly.    | $\Theta(k + n)$ | $\Theta(k + n)$ |
| Radix Sort         | Sort on each “digit” using stable sort            | $\Theta(d(n + k))$ | $\Theta(d(n + k))$ |
| Bucket Sort        | Divide range into buckets, place accordingly      | $\Theta(n^2)$ | $\Theta(n)$ |
This Course

• Sorting
  – A necessary tool for managing and navigating a variety of datasets.
  – A great context in which to learn computer science
    • Analyze Complexity
    • Prove Correctness

• Data Structures
  – Explore variety of structures beyond arrays and linked lists.
  – How you store data affects how easily you can manage and navigate it.

• Optimization Problems
  – Learn sophisticated methods that are alternates to exponential brute force approach.

• Graph Problems
  – Applicable to wide range of problems (many seemingly unrelated problems map to a graph problem).
  – Another great context in which to learn computer science.