Chapter 8: Non-Comparison-Based Sorting - Lower Bounds

- All of the sorting algorithms seen so far (merge, insertion, quick) are based on comparing elements
  - Such sorts are called *comparison sorts*
- Comparison sorts can be represented by a *decision tree*
  - Decision tree is binary
  - Each node represents a comparison of two elements
  - Nodes are labeled $i:j$, indicating element $i$ is being compared to element $j$, where $1 \leq i, j \leq n$, and $n$ is the number of elements being sorted
  - Leaves represent the sorted results
    * Since sorting could result in any permutation of the $n$ elements, there are $n!$ leaves in the tree representing all possible results
  - A sort of a specific input is represented as a path from the root of the tree to a leaf
  - Will consider only $\leq$ comparisons
  - Given comparison $a_i \leq a_j$ at a node, the left branch represents $a_i \leq a_j$, the right branch $a_i > a_j$

- A sorting algorithm should be able to generate all permutations of the input
- Each leaf is *reachable* from the root as a result of a sort
- The length of a simple path from the root to a leaf represents the worst case number of comparisons required to perform a sort
- The upper bound on a tree’s height represents a lower bound for worst case sorting (by comparisons)
• Theorem (8.1): Any comparison sort requires $\Omega(n \lg n)$ comparisons in the worst case
  
  – Proof:
  * Consider a decision tree of height $h$ with $l$ reachable leaves that repre-
sents the results of a comparison sort of $n$ elements
  * Each of the $n!$ permutations of the $n$ elements appear at the leaf level, so $n! \leq l$
  * A binary tree of height $h$ has $\leq 2^h$ leaves, so
    $$n! \leq l \leq 2^h$$
  * Taking the lg,
    $$h \geq \lg(n!) = n\lg n$$
  * Therefore, the number of comparisons $\in \Omega(n \lg n)$

• The import of this theorem is that no comparison sort can do better than $\in \Omega(n \lg n)$ in the worst case
  
  – However, sorts that are based on techniques other than comparisons do not have this limitation
Chapter 8: Non-Comparison-Based Sorting - Counting Sort

• Input is an array $A[1\ldots n]$ of elements $0 \leq x_i \leq k$

• Two auxiliary arrays are used:
  1. $C[0\ldots k]$ for intermediate processing
  2. $B[1\ldots n]$ for final results

• General technique:
  - Count how many values $x_i$ there are in $A$, where $0 \leq x_i \leq k$
  - Maintain these counts in $C[i]$
  - $C$ will then be adjusted so that the value in $C[i]$ represents the number of elements $\leq x_i$ in $A$
  - This will allow inserting $x_i$ directly into its proper location in $B$

• Algorithm:

```java
Counting-Sort (A, B, k) {
  C = new array[0 .. k]
  for (i = 0 to k)
    C[i] = 0
  for (j = 1 to A.length)
    C[A[j]] = C[A[j]]++
  //C[i] now holds the number of elements whose value is i
  for (i = 1 to k)
    C[i] = C[i] + C[i - 1]
  //C[i] now holds number of elements $\leq i$
  for (j = A.length down to 1) {
    B[C[A[j]]] = A[j]
    C[A[j]]--
  }
}
```
Chapter 8: Non-Comparison-Based Sorting - Counting Sort (2)

• Analysis:
  – Lines 2 - 3: $\Theta(k)$
  – Lines 4 - 5: $\Theta(n)$
  – Lines 7 - 8: $\Theta(k)$
  – Lines 10 - 12: $\Theta(n)$
  – So $T(n) \in \Theta(k + n)$
  – In practice, $k \in O(n)$, so $T(n) \in \Theta(n)$

• This is a stable algorithm: Relative order of elements of the same value remains unchanged by the sort

• This sort is able to achieve better than $\Omega(n \log n)$ time because it does not rely on element comparisons
Chapter 8: Non-Comparison-Based Sorting - Radix Sort

• Input: Array \( A[1 \ldots n] \) of \( d \)-digit numbers

• Technique:
  – \( d \) passes will be made through the input
  – On each pass, the keys will be sorted \textit{based on a single digit}, working from least significant to most significant
  – The sort used must be stable

• Algorithm:

\[
\text{Radix-Sort} \ (A, d) \\
\quad \{
\quad \text{for (i = 1 to d)} \\
\quad \quad \text{call a stable sort to sort A based on digit i}
\quad \}
\]

• Lemma (8.3): Given \( n \) \( d \)-digit numbers in which the digits may assume up to \( k \) possible values, \textit{Radix-Sort} correctly sorts in \( \Theta(d(n + k)) \) time if the stable sort takes \( \Theta(n + k) \) time

  – Proof:
    * Run time depends on the sort
    * Lines 1 - 2 perform \( d \) iterations
    * If \textit{Counting Sort} is used as the stable sort, the sort \( \in \Theta(n + k) \)
    * Therefore \( T(n) \in \Theta(d(n + k)) \)

• Lemma (8.4): Given \( n \) \( b \)-bit numbers and any positive integer \( r \leq b \), \textit{Radix-Sort} correctly sorts in \( \Theta((b/r)(n + 2^r)) \) time if the stable sort takes \( \Theta(n + k) \) time for inputs in the range \([0, k]\)

  – Proof:
    * For \( r \leq b \), can consider each key as having \( d = \lfloor b/r \rfloor \) digits of \( r \) bits
    * Each digit is an integer in range \( 0 \ldots 2^r - 1 \)
    * \textit{Counting Sort} can be used with \( k = 2^r \)
    * Each pass of the sort takes \( \Theta(n + k) = \Theta(n + 2^r) \) time
    * With \( d \) passes, have \( \Theta(d(n + 2^r)) = \Theta((b/r)(n + 2^r)) \)
Chapter 8: Non-Comparison-Based Sorting - Radix Sort (2)

- To make the algorithm most efficient, want the value of $r$ that minimizes $(b/r)(n + 2^r)$
  - If $b < \lfloor \log n \rfloor$, then for any $r \leq b$, $(n + 2^r) = \Theta(n)$ which is asymptotically optimal
  - If $b \geq \lfloor \log n \rfloor$, then $r = \lfloor \log n \rfloor$ gives the best time within a constant factor
    * This gives run time of $\Theta((bn)/(\log n))$
    * As $r$ increases, $2^r$ grows faster than $r$ and run time $\in \Omega((bn)/(\log n))$
    * If $r$ decreases $< \lfloor \log n \rfloor$, $b/r$ increases and $n + 2^r$ remains $\Theta(n)$
Chapter 8: Non-Comparison-Based Sorting - Bucket Sort

- Input: Array $A[1 \ldots n]$ of elements in range $[0, 1)$
  - Elements assumed to be uniformly and independently distributed over this range

- An auxiliary array is used:
  - $B[0 \ldots n - 1]$
  - $B$ is an array of linked lists into which elements of $A$ are inserted
  - Each $B[i]$ is called a *bucket*
  - $B$ represents $n$ equally-spaced intervals over $[0, 1)$
  - In the ideal case, only one element from $A$ will occupy each bucket

- General technique:
  - Put elements from $A$ into their appropriate buckets
  - Sort the individual buckets
  - Concatenate the buckets

- Algorithm:

```java
Bucket-Sort (A) {
1     n = A.length
2     B = new array[0 .. n - 1]
3     for (i = 0 to n - 1)
4         B[i] = 0
5     for (i = 1 to n)
6         insert A[i] into B[floor(n * A[i])]
7     for (i = 0 to n - 1)
8         Insertion-Sort(B[i])
9     concatenate B[0], B[1], ..., B[n - 1]
}
```
• Proof of correctness:
  – Since \( \lfloor nA[i] \rfloor \leq \lfloor nA[j] \rfloor \), \( A[i] \) goes in the same bucket as \( A[j] \) or in a lower one
  – If the same bucket, sorting (line 8) will put them in proper order
  – If in different buckets, concatenation (line 9) will put them in proper order

• Analysis:
  – Loops in lines 3 - 6 \( \in \mathcal{O}(n) \)
  – Loop lines 7 - 8
    * Let \( n_i \) be a random variable representing the number of keys in \( B[i] \)
    * Insertion sort \( \in \mathcal{O}(n^2) \)
    * Therefore
      \[
      T(n) = \Theta(n) + \sum_{i=0}^{n-1} \mathcal{O}(n_i^2)
      \]
  – For the average case
    \[
    E[T(n)] = E\left[\Theta(n) + \sum_{i=0}^{n-1} \mathcal{O}(n_i^2)\right]
    \]
    \[
    = \Theta(n) + \sum_{i=0}^{n-1} E[\mathcal{O}(n_i^2)]
    \]
    \[
    = \Theta(n) + \sum_{i=0}^{n-1} \mathcal{O}(E[n_i^2])
    \]
    \[
    = \Theta(n) + \sum_{i=0}^{n-1} \mathcal{O}(2 - \frac{1}{n})
    \]
    * \( E[n_i^2] = 2 - \frac{1}{n} \) for \( i = 0, 1, \ldots, n - 1 \)
Chapter 8: Non-Comparison-Based Sorting - Bucket Sort (3)

* Proof:
  · Let $X_{ij} = I \{A[j] \text{ falls in bucket } i\}, 0 \leq i \leq n-1, 1 \leq j \leq n$
  · Then $n_i = \sum_{j=1}^{n} X_{ij}$
  · Substituting in the above
    
    $E[n_i^2] = E \left[ \left( \sum_{j=1}^{n} (X_{ij}) \right)^2 \right]$

    
    $= E \left[ \sum_{j=1}^{n} \sum_{k=1}^{n} (X_{ij}X_{ik}) \right]$

    
    $= E \left[ \sum_{k=1}^{n} X_{ij}^2 + \sum_{j=1}^{n} \sum_{k=1}^{n} (X_{ij}X_{ik}) \right]$

    
    $= \sum_{k=1}^{n} E[X_{ij}^2] + \sum_{j=1}^{n} \sum_{k=1}^{n} E[X_{ij}X_{ik}]$

    
    · $X_{ij} = 1$ with probability $1/n$, so
      
      $E[X_{ij}^2] = 1^2 \cdot \frac{1}{n} + 0^2 \cdot \left(1 - \frac{1}{n}\right) = \frac{1}{n}$

    · When $k \neq j$, $X_{ij}$ and $X_{ik}$ are independent, so
      
      $E[X_{ij}X_{ik}] = E[X_{ij}]E[X_{ik}] = \frac{1}{n} \cdot \frac{1}{n} = \frac{1}{n^2}$

    · Substituting in the above
      
      $E[n_i^2] = \sum_{j=1}^{n} \frac{1}{n} + \sum_{k=1}^{n} \frac{1}{n^2}$

      
      $= n \cdot \frac{1}{n} + n(n-1) \cdot \frac{1}{n^2}$

      
      $= 1 + \frac{n-1}{n}$

      
      $= 2 - \frac{1}{n}$

    * Therefore
      
      $T_{avg}(n) = \Theta(n) + n \cdot O \left(2 - \frac{1}{n}\right) = \Theta(n)$