Design and Analysis of Algorithms

CSE 5311
Lecture 8  Sorting in Linear Time

Junzhou Huang, Ph.D.
Department of Computer Science and Engineering
Sorting So Far

• Insertion sort:
  – Easy to code
  – Fast on small inputs (less than ~50 elements)
  – Fast on nearly-sorted inputs
  – $O(n^2)$ worst case
  – $O(n^2)$ average (equally-likely inputs) case
  – $O(n^2)$ reverse-sorted case

• Merge sort:
  – Divide-and-conquer:
    ➢ Split array in half
    ➢ Recursively sort subarrays
    ➢ Linear-time merge step
  – $O(n \lg n)$ worst case
Sorting So Far

• **Heap sort:**
  - Uses the very useful heap data structure
    - Complete binary tree
    - Heap property: parent key > children’s keys
  - \( O(n \lg n) \) worst case
  - Sorts in place
  - Fair amount of shuffling memory around

• **Quick sort:**
  - Divide-and-conquer:
    - Partition array into two subarrays, recursively sort
    - All of first subarray < all of second subarray
    - No merge step needed!
  - \( O(n \lg n) \) average case
  - Fast in practice
  - \( O(n^2) \) worst case
    - Naïve implementation: worst case on sorted input
    - Address this with randomized quicksort
**How Fast Can We Sort?**

- **Lower bound**
  - Prove a Lower Bound for *any comparison based algorithm* for the Sorting Problem
  - *How?* Decision trees help us.

- **Observation**: sorting algorithms so far are *comparison sorts*
  - The only operation used to gain ordering information about a sequence is the pairwise comparison of two elements
  - Theorem: all comparison sorts are $\Omega(n \log n)$
    - A comparison sort must do $O(n)$ comparisons (why?)
    - What about the gap between $O(n)$ and $O(n \log n)$
Decision-tree Example

Sort \( \langle a_1, a_2, \ldots, a_n \rangle \)

Each internal node is labeled \( i:j \) for \( i, j \in \{1, 2, \ldots, n\} \).
- The left subtree shows subsequent comparisons if \( a_i \leq a_j \).
- The right subtree shows subsequent comparisons if \( a_i \geq a_j \).
Each internal node is labeled $i:j$ for $i, j \in \{1, 2, \ldots, n\}$.

- The left subtree shows subsequent comparisons if $a_i \leq a_j$.
- The right subtree shows subsequent comparisons if $a_i \geq a_j$.

Decision-tree Example

Sort $\langle a_1, a_2, a_3 \rangle = \langle 9, 4, 6 \rangle$:
Decision-tree Example

Sort $\langle a_1, a_2, a_3 \rangle = \langle 9, 4, 6 \rangle$:

Each internal node is labeled $i:j$ for $i, j \in \{1, 2, \ldots, n\}$.

- The left subtree shows subsequent comparisons if $a_i \leq a_j$.
- The right subtree shows subsequent comparisons if $a_i \geq a_j$. 
Each internal node is labeled $i:j$ for $i, j \in \{1, 2, \ldots, n\}$.

- The left subtree shows subsequent comparisons if $a_i \leq a_j$.
- The right subtree shows subsequent comparisons if $a_i \geq a_j$.
Decision-tree Example

Sort $\langle a_1, a_2, a_3 \rangle = \langle 9, 4, 6 \rangle$:

Each leaf contains a permutation $\langle \pi(1), \pi(2), \ldots, \pi(n) \rangle$ to indicate that the ordering $a_{\pi(1)} \leq a_{\pi(2)} \leq \cdots \leq a_{\pi(n)}$ has been established.
A decision tree can model the execution of any comparison sort:

- One tree for each input size $n$.
- View the algorithm as splitting whenever it compares two elements.
- The tree contains the comparisons along all possible instruction traces.
- The running time of the algorithm = the length of the path taken.
- Worst-case running time = height of tree.
How?

Any comparison sort can be turned into a Decision tree

class InsertionSortAlgorithm {
    for (int i = 1; i < a.length; i++) {
        int j = i;
        while ((j > 0) && (a[j-1] > a[i])) {
            a[j] = a[j-1];
            j--;
        }
        a[j] = B;
    }
}
Lower Bound for Decision-tree Sorting

**Theorem.** Any decision tree that can sort \( n \) elements must have height \( \Omega(n \lg n) \).

**Proof.** The tree must contain \( \geq n! \) leaves, since there are \( n! \) possible permutations. A height-\( h \) binary tree has \( \leq 2^h \) leaves. Thus, \( n! \leq 2^h \).

\[
\begin{align*}
\therefore \quad h & \geq \lg(n!)
\geq \lg \left(\left(\frac{n}{e}\right)^n\right)
= n \lg n - n \lg e
= \Omega(n \lg n).
\end{align*}
\]

\( \lg \) is mono. increasing  
(Stirling’s formula)

\[ n \log n - n < \log(n!) < n \log n \]
Decision Tree

• Decision trees provide an abstraction of comparison sorts
  – A decision tree represents the comparisons made by a comparison sort.
    Every thing else ignored
  – What do the leaves represent?
  – How many leaves must there be?
• Decision trees can model comparison sorts. For a given algorithm:
  – One tree for each $n$
  – Tree paths are all possible execution traces
    – What’s the longest path in a decision tree for insertion sort? For merge sort?

• What is the asymptotic height of any decision tree for sorting $n$ elements?
• Answer: $\Omega(n \lg n)$ (now let’s prove it…)
Lower Bound For Comparison Sorting

- **Theorem:** Any decision tree that sorts \( n \) elements has height \( \Omega(n \log n) \)
- What’s the minimum # of leaves?
- What’s the maximum # of leaves of a binary tree of height \( h \)?
- Clearly the minimum # of leaves is less than or equal to the maximum # of leaves

So we have \( n! \leq 2^h \); Taking logarithms: \( \log (n!) \leq h \)

- Stirling’s approximation tells us: \( n! > \left(\frac{n}{e}\right)^n \)

Thus \( h \geq \log \left(\frac{n}{e}\right)^n = n \log n - n \log e = \Omega(n \log n) \)

The minimum height of a decision tree is \( \Omega(n \log n) \)
Lower Bound For Comparison Sorting

• Thus the time to comparison sort \( n \) elements is \( \Omega(n \lg n) \)

• **Corollary**: Heapsort and Mergesort are asymptotically optimal comparison sorts

• But the name of this lecture is “Sorting in linear time”!
  
  – *How can we do better than \( \Omega(n \lg n) \)?*
Sorting In Linear Time

• Counting sort
  – No comparisons between elements!
  – But... depends on assumption about the numbers being sorted
    ➢ We assume numbers are in the range $1\ldots k$
  – The algorithm:
    ➢ Input: $A[1..n]$, where $A[j] \in \{1, 2, 3, \ldots, k\}$
    ➢ Output: $B[1..n]$, sorted (notice: not sorting in place)
    ➢ Also: Array $C[1..k]$ for auxiliary storage
Counting Sort

1 CountingSort(A, B, k)
2     for i=1 to k
3         C[i] = 0;
4     for j=1 to n
5         C[A[j]] += 1;
6     for i=2 to k
7         C[i] = C[i] + C[i-1];
8     for j=n downto 1
9         B[C[A[j]]] = A[j];
10         C[A[j]] -= 1;

Work through example: A={4 1 3 4 3}, k = 4
Counting Sort

1. `CountingSort(A, B, k)`
   2. for `i=1` to `k` Takes time $O(k)$
      3. `C[i] = 0;`
   4. for `j=1` to `n` Takes time $O(n)$
      5. `C[A[j]] += 1;`
   6. for `i=2` to `k`
      7. `C[i] = C[i] + C[i-1];`
   8. for `j=n` downto `1`
   10. `C[A[j]] -= 1;`

What will be the running time?
Counting-sort Example

A: 4 1 3 4 3

B: 

C: 1 2 3 4

Dept. CSE, UT Arlington  CSE5311 Design and Analysis of Algorithms
Loop 1

\[
\text{for } i \leftarrow 1 \text{ to } k \\
\text{do } C[i] \leftarrow 0
\]
Loop 2

\[
\begin{array}{cccccc}
A: & 4 & 1 & 3 & 4 & 3 \\
B: & & & & & \\
C: & 1 & 2 & 3 & 4 & 1 \\
\end{array}
\]

\[
\text{for } j \leftarrow 1 \text{ to } n \\
\text{do } C[A[j]] \leftarrow C[A[j]] + 1 \\
\text{ } i \} | \\
\]

\[\triangleright C[i] = |\{\text{key} = \]

Dept. CSE, UT Arlington

CSE5311 Design and Analysis of Algorithms

21
Loop 2

\[
\begin{align*}
A: & \\
& \begin{array}{cccccc}
1 & 2 & 3 & 4 & 5 \\
4 & 1 & 3 & 4 & 3 \\
\end{array} \\
B: & \\
C: & \\
& \begin{array}{cccccc}
1 & 2 & 3 & 4 \\
1 & 0 & 0 & 1 \\
\end{array}
\end{align*}
\]

\[
\text{for } j \leftarrow 1 \text{ to } n \\
do \quad C[A[j]] \leftarrow C[A[j]] + 1 \\
\text{\triangleright } C[i] = |\{\text{key } = i\}|
\]
Loop 2

\[
\begin{array}{c}
A: \\
\begin{array}{cccccc}
1 & 2 & 3 & 4 & 5 \\
4 & 1 & 3 & 4 & 3
\end{array} \\
B: \\

C: \\
\begin{array}{cccccc}
1 & 2 & 3 & 4 \\
1 & 0 & 1 & 1
\end{array}
\end{array}
\]

\[
\text{for } j \leftarrow 1 \text{ to } n \\
\quad \text{do } C[A[j]] \leftarrow C[A[j]] + 1 \\
\quad \quad i \}} = |\{\text{key} = \]

Dept. CSE, UT Arlington

CSE5311 Design and Analysis of Algorithms
Loop 2

\[\begin{array}{llllll}
A: & 4 & 1 & 3 & 4 & 3 \\
B: &  &  &  &  &  \\
C: & 1 & 0 & 1 & 2 &  \\
\end{array}\]

\[\text{for } j \leftarrow 1 \text{ to } n\]
\[\text{do } C[A[j]] \leftarrow C[A[j]] + 1\]
\[\triangleright C[i] = |\{\text{key } = i\}|\]
Loop 2

\[
\begin{align*}
&\text{for } j \leftarrow 1 \text{ to } n \\
&\quad \text{do } C[A[j]] \leftarrow C[A[j]] + 1 \\
&\quad i \leftarrow C[i] = |\{\text{key } = i\}| \\
\end{align*}
\]
Loop 3

for $i \leftarrow 2$ to $k$

\[ C[i] \leftarrow C[i] + C[i-1] \quad \triangleright \quad C[i] = |\{\text{key} \leq i\}| \]
Loop 3

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>4</td>
<td>1</td>
<td>3</td>
<td>4</td>
<td>3</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>C</td>
<td>1</td>
<td>0</td>
<td>2</td>
<td>2</td>
</tr>
</tbody>
</table>

for $i \leftarrow 2$ to $k$

do $C[i] \leftarrow C[i] + C[i-1]$  \triangleright  $C[i] = |\{\text{key} \leq i\}|$

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>$B'$</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>1</th>
<th>3</th>
<th>2</th>
</tr>
</thead>
<tbody>
<tr>
<td>$C'$</td>
<td>1</td>
<td>1</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Loop 3

\[
\begin{align*}
A: & \begin{array}{ccccc}
1 & 2 & 3 & 4 & 5 \\
4 & 1 & 3 & 4 & 3 \\
\end{array} & C: & \begin{array}{ccccc}
1 & 2 & 3 & 4 \\
1 & 0 & 2 & 2 \\
\end{array} \\
B: & \begin{array}{cccc}
\hline \\
\hline \\
\hline \\
\hline \\
\end{array} & C': & \begin{array}{ccccc}
1 & 1 & 3 & 5 \\
1 & 1 & 3 & 5 \\
\end{array} \\
\end{align*}
\]

**for** \(i \leftarrow 2\) **to** \(k\) \\
**do** \(C[i] \leftarrow C[i] + C[i-1]\) \(\triangleright C[i] = |\{\text{key} \leq i\}|\)
for $j \leftarrow n$ downto 1

\[ B[C[A[j]]] \leftarrow A[j] \]
\[ C[A[j]] \leftarrow C[A[j]] - 1 \]
Loop 4

\[
\begin{align*}
A & : 4 \ 1 \ 3 \ 4 \ 3 \\
B & : \quad \quad \quad 3 \ 4 \\
C & : 1 \ 1 \ 2 \ 5 \\
C' & : 1 \ 1 \ 2 \ 4
\end{align*}
\]

\textbf{for } j \leftarrow n \textbf{ downto } 1 \\
\textbf{do } B[C[A[j]]] \leftarrow A[j] \\
\textbf{do } C[A[j]] \leftarrow C[A[j]] - 1
Loop 4

for $j \leftarrow n$ downto 1

\[ B[C[A[j]]] \leftarrow A[j] \]
\[ C[A[j]] \leftarrow C[A[j]] - 1 \]
Loop 4

\[
\begin{array}{ccccc}
A: & 4 & 1 & 3 & 4 & 3 \\
B: & 1 & 3 & 3 & 4 \\
C: & 1 & 1 & 1 & 4 \\
C': & 0 & 1 & 1 & 4 \\
\end{array}
\]

for \( j \gets n \) downto 1

- \( B[C[A[j]]] \gets A[j] \)
- \( C[A[j]] \gets C[A[j]] - 1 \)
Loop 4

for $j \leftarrow n$ downto 1
    do $B[C[A[j]]] \leftarrow A[j]$
        $C[A[j]] \leftarrow C[A[j]] - 1$
Analysis

\[\Theta(k) \quad \text{for } i \leftarrow 1 \text{ to } k \]
\[\text{do } C[i] \leftarrow 0\]

\[\Theta(n) \quad \text{for } j \leftarrow 1 \text{ to } n \]
\[\text{do } C[A[j]] \leftarrow C[A[j]] + 1\]

\[\Theta(k) \quad \text{for } i \leftarrow 2 \text{ to } k \]
\[\text{do } C[i] \leftarrow C[i] + C[i-1]\]

\[\Theta(n) \quad \text{for } j \leftarrow n \text{ downto } 1\]
\[\text{do } B[C[A[j]]] \leftarrow A[j]\]
\[C[A[j]] \leftarrow C[A[j]] - 1\]

\[\Theta(n + k)\]
Counting Sort

- **Total time:** $O(n + k)$
  - Usually, $k = O(n)$
  - Thus counting sort runs in $O(n)$ time

- **But sorting is $\Omega(n \lg n)$!**
  - No contradiction--this is not a comparison sort (in fact, there are no comparisons at all!)
  - Notice that this algorithm is **stable**

- Cool! Why don’t we always use counting sort?
- Because it depends on range $k$ of elements
- Could we use counting sort to sort 32 bit integers? Why or why not?
- Answer: no, $k$ too large ($2^{32} = 4,294,967,296$)
Stable Sorting

Counting sort is a \textit{stable} sort: it preserves the input order among equal elements.

\begin{itemize}
\item[A:] \begin{tabular}{cccccc}
4 & 1 & 3 & 4 & 3 \\
\end{tabular}
\item[B:] \begin{tabular}{cccccc}
1 & 3 & 3 & 4 & 4 \\
\end{tabular}
\end{itemize}

\textbf{Exercise:} What other sorts have this property?
Radix Sort

- Intuitively, you might sort on the **most significant digit**, then the second msd, etc.

- **Problem:** lots of intermediate piles of cards (read: scratch arrays) to keep track of

- **Key idea:** sort the *least* significant digit first

  RadixSort(A, d)
  
  for i=1 to d
  
  StableSort(A) on digit i
  
  - Example: Fig 9.3
Radix Sort

• *Can we prove it will work?*

• **Sketch of an inductive argument** (induction on the number of passes):
  - Assume lower-order digits \{j: j<i\} are sorted
  - Show that sorting next digit i leaves array correctly sorted
    - If two digits at position i are different, ordering numbers by that digit is correct (lower-order digits irrelevant)
    - If they are the same, numbers are already sorted on the lower-order digits. Since we use a stable sort, the numbers stay in the right order
Radix Sort

• **What sort will we use to sort on digits?**
• **Counting sort is obvious choice:**
  – Sort $n$ numbers on digits that range from $1..k$
  – Time: $O(n + k)$
• **Each pass over $n$ numbers with $d$ digits takes time $O(n+k)$, so total time $O(dn+dk)$**
  – When $d$ is constant and $k=O(n)$, takes $O(n)$ time
• **How many bits in a computer word?**
Radix Sort

- **Problem:** sort 1 million 64-bit numbers
  - Treat as four-digit radix $2^{16}$ numbers
  - Can sort in just four passes with radix sort!

- **Compares well with typical $O(n \lg n)$ comparison sort**
  - Requires approximate $\log n = 20$ operations per number being sorted

- **So why would we ever use anything but radix sort?**

- **In general, radix sort based on counting sort is**
  - Fast, Asymptotically fast (i.e., $O(n)$)
  - Simple to code
  - A good choice

- **To think about:** *Can radix sort be used on floating-point numbers?*
Operation of Radix Sort

3 2 9
4 5 7
6 5 7
8 3 9
4 3 6
7 2 0
3 5 5
7 2 0
3 5 5
3 2 9
4 3 6
4 3 6
8 3 9
3 5 5
6 5 7
4 5 7
7 2 0
8 3 9
6 5 7
8 3 9
Correctness of Radix Sort

**Induction on digit position**

- Assume that the numbers are sorted by their low-order $t - 1$ digits.

- Sort on digit $t$
Correctness of Radix Sort

*Induction on digit position*

- Assume that the numbers are sorted by their low-order $t - 1$ digits.

- Sort on digit $t$
  - Two numbers that differ in digit $t$ are correctly sorted.
Correctness of Radix Sort

*Induction on digit position*

- Assume that the numbers are sorted by their low-order \( t-1 \) digits.

- Sort on digit \( t \)
  - Two numbers that differ in digit \( t \) are correctly sorted.
  - Two numbers equal in digit \( t \) are put in the same order as the input \( \implies \) correct order.
Analysis of Radix Sort

• Assume counting sort is the auxiliary stable sort.
• Sort \( n \) computer words of \( b \) bits each.
• Each word can be viewed as having \( b/r \) base-\(2^r\) digits.

**Example:** 32-bit word

\[ \begin{array}{cccc}
8 & 8 & 8 & 8 \\
\end{array} \]

\( r = 8 \quad b/r = 4 \) passes of counting sort on base-\(2^8\) digits; or \( r = 16 \quad b/r = 2 \) passes of counting sort on base-\(2^{16}\) digits.

*How many passes should we make?*
Recall: Counting sort takes $\Theta(n + k)$ time to sort $n$ numbers in the range from 0 to $k - 1$.

If each $b$-bit word is broken into $r$-bit pieces, each pass of counting sort takes $\Theta(n + 2^r)$ time. Since there are $b/r$ passes, we have

$$\Theta\left(\frac{b}{r}n + 2^r\right)$$

Choose $r$ to minimize $T(n, b)$:

- Increasing $r$ means fewer passes, but as $r >> \lg n$, the time grows exponentially.
Choosing $r$

Minimize $T(n, b)$ by differentiating and setting to 0.

Or, just observe that we don’t want $2^r > n$, and there’s no harm asymptotically in choosing $r$ as large as possible subject to this constraint.

Choosing $r = \lg n$ implies $T(n, b) = \Theta(b \frac{n}{\lg n})$.

• For numbers in the range from 0 to $n^d - 1$, we have $b = d \lg n \Rightarrow$ radix sort runs in $\Theta(dn)$ time.
Bucket Sort

• **Assumption: uniform distribution**
  – Input numbers are *uniformly distributed* in [0,1).
  – Suppose input size is \( n \).

• **Idea:**
  – Divide [0,1) into \( n \) equal-sized subintervals (buckets).
  – Distribute \( n \) numbers into buckets
  – Expect that each bucket contains few numbers.
  – Sort numbers in each bucket (insertion sort as default).
  – Then go through buckets in order, listing elements,
BUCKET-SORT(A)

1. \( n \leftarrow \text{length}[A] \)
2. \( \text{for } i \leftarrow 1 \text{ to } n \)
3. \( \text{do insert } A[i] \text{ into bucket } B[nA[i]] \)
4. \( \text{for } i \leftarrow 0 \text{ to } n-1 \)
5. \( \text{do sort bucket } B[i] \text{ using insertion sort} \)
6. \( \text{Concatenate bucket } B[0], B[1], \ldots, B[n-1] \)
Example of BUCKET-SORT

![Diagram of BUCKET-SORT](image)

**Figure 8.4** The operation of BUCKET-SORT. (a) The input array $A[1..10]$. (b) The array $B[0..9]$ of sorted lists (buckets) after line 5 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 $B[0], B[1], \ldots, B[9]$. 
Analysis of BUCKET-SORT(A)

1. \( n \leftarrow \text{length}[A] \) \( \Omega(1) \)
2. \( \text{for } i \leftarrow 1 \text{ to } n \) \( O(n) \)
3. \( \text{do insert } A[i] \text{ into bucket } B[\lfloor nA[i] \rfloor] \) \( \Omega(1) \) (i.e. total \( O(n) \))
4. \( \text{for } i \leftarrow 0 \text{ to } n-1 \) \( O(n) \)
5. \( \text{do sort bucket } B[i] \text{ with insertion sort } O(n_i^2) \) \( \sum_{i=0}^{n-1} O(n_i^2) \)
6. \( \text{Concatenate bucket } B[0],B[1],\ldots,B[n-1] \) \( O(n) \)

Where \( n_i \) is the size of bucket \( B[i] \).

Thus \( T(n) = \Theta(n) + \sum_{i=0}^{n-1} O(n_i^2) \)
\[ = \Theta(n) + n \cdot O(2^{-1/n}) = \Theta(n) \]
Analysis of BUCKET-SORT(A)

Time: 
\[ T(n) = \Theta(n) + \sum_{i=0}^{n-1} O(n_i^2) \] 
\( (n_i: \text{number of elements in } i^{th} \text{ bucket}) \)

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

\[ = \Theta(n) + \sum_{i=0}^{n-1} E[O(n_i^2)] \quad \text{(linearity of expectation)} \]

\[ = \Theta(n) + \sum_{i=0}^{n-1} O(E[n_i^2]) \quad \text{([E[aX]] = aE[X])} \]

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

\[ = \Theta(n) \]