Sorting in Linear Time

Recitation: January 26, 2018

CS430: Introduction to Algorithms

Linlin Chen
lchen96@hawk.iit.edu
### So far...

<table>
<thead>
<tr>
<th>Name</th>
<th>Best</th>
<th>Average</th>
<th>Worst</th>
</tr>
</thead>
<tbody>
<tr>
<td>Insertion Sort</td>
<td>$n$</td>
<td>$n^2$</td>
<td>$n^2$</td>
</tr>
<tr>
<td>Heap Sort</td>
<td>$n \log n$</td>
<td>$n \log n$</td>
<td>$n \log n$</td>
</tr>
<tr>
<td>Quick Sort</td>
<td>$n \log n$</td>
<td>$n \log n$</td>
<td>$n^2$</td>
</tr>
<tr>
<td>Merge Sort</td>
<td>$n \log n$</td>
<td>$n \log n$</td>
<td>$n \log n$</td>
</tr>
</tbody>
</table>
So far...

<table>
<thead>
<tr>
<th>Name</th>
<th>Best</th>
<th>Average</th>
<th>Worst</th>
</tr>
</thead>
<tbody>
<tr>
<td>Insertion Sort</td>
<td>$n$</td>
<td>$n^2$</td>
<td>$n^2$</td>
</tr>
<tr>
<td>Heap Sort</td>
<td>$n \log n$</td>
<td>$n \log n$</td>
<td>$n \log n$</td>
</tr>
<tr>
<td>Quick Sort</td>
<td>$n \log n$</td>
<td>$n \log n$</td>
<td>$n^2$</td>
</tr>
<tr>
<td>Merge Sort</td>
<td>$n \log n$</td>
<td>$n \log n$</td>
<td>$n \log n$</td>
</tr>
</tbody>
</table>

Sorting algorithms so far determine the sorted order based only on comparisons between the input elements. These are called comparison sorts.
Lower bounds of Comparison Sorts

**Theorem 8.1**
Any comparison sort algorithm requires $\Omega(n \lg n)$ comparisons in the worst case.

**Proof** From the preceding discussion, it suffices to determine the height of a decision tree in which each permutation appears as a reachable leaf. Consider a decision tree of height $h$ with $l$ reachable leaves corresponding to a comparison sort on $n$ elements. Because each of the $n!$ permutations of the input appears as some leaf, we have $n! \leq l$. Since a binary tree of height $h$ has no more than $2^h$ leaves, we have

$$n! \leq l \leq 2^h,$$

which, by taking logarithms, implies

$$h \geq \lg(n!) \quad \text{(since the lg function is monotonically increasing)}$$

$$= \Omega(n \lg n) \quad \text{(by equation (3.19))}.$$
Non-Comparison Sorts?

Can we avoid to compare for sorting?

Insights:

1. If we can know the position of each element in sorted array?

2. Maybe we can compare numbers chunk by chunk? Like compare year first, then month, then day?

3. Or maybe learn from some idea of histograms?
Intuition:
If we can get the position of each element in the sorted array in some way, then sorting becomes just putting each element in its corresponding spot.

Assumption:
Each of the $n$ input elements is an integer in the range 0 to $k$, for some integer $k$. Counting sort runs in linear time ($\Theta(n)$) only when $k = O(n)$
How it works

• Uses two auxiliary arrays:
  − $B[1, ..., n]$: store the sorted elements.
  − $C[0, ..., k]$: stores the number of items less than $i$, for $0 \leq i \leq k$. 
Algorithm

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
Time Complexity

\[
\text{COUNTING-SORT}(A, B, k)
\]

1. let \( C[0..k] \) be a new array
2. \( \Theta(k) \)
   - for \( i = 0 \) to \( k \)
   - \( C[i] = 0 \)
3. \( \Theta(n) \)
   - for \( j = 1 \) to \( A.length \)
   - \( C[A[j]] = C[A[j]] + 1 \)
   - // \( C[i] \) now contains the number of elements equal to \( i \).
4. \( \Theta(k) \)
   - for \( i = 1 \) to \( k \)
   - \( C[i] = C[i] + C[i - 1] \)
   - // \( C[i] \) now contains the number of elements less than or equal to \( i \).
5. \( \Theta(n) \)
   - for \( j = A.length \) down to 1
   - \( B[C[A[j]]] = A[j] \)
   - \( C[A[j]] = C[A[j]] - 1 \)

Total running complexity: \( \Theta(k + n) = \Theta(n) \) when \( k = O(n) \)
Example

(a) $A = \begin{array}{cccccccc}
1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 \\
2 & 5 & 3 & 0 & 2 & 3 & 0 & 3 \\
0 & 1 & 2 & 3 & 4 & 5 \\
2 & 0 & 2 & 3 & 0 & 1 \\
\end{array}$

(b) $C = \begin{array}{cccc}
2 & 2 & 4 & 7 \\
7 & 7 & 8 \\
\end{array}$

(c) $B = \begin{array}{cccccccc}
1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 \\
0 & 1 & 2 & 3 & 4 & 5 \\
2 & 2 & 4 & 6 & 7 & 8 \\
\end{array}$

(d) $B = \begin{array}{cccccccc}
1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 \\
0 & 1 & 2 & 3 & 4 & 5 \\
1 & 2 & 4 & 6 & 7 & 8 \\
\end{array}$

(e) $C = \begin{array}{cccc}
1 & 2 & 4 & 5 \\
7 & 8 \\
\end{array}$

(f) $B = \begin{array}{cccccccc}
1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 \\
0 & 0 & 2 & 2 & 3 & 3 & 3 & 5 \\
\end{array}$
Property

• Counting sort is good for sorting integers in a narrow range \( k = O(n) \)
• It assumes the input numbers are in range 0, 1, ..., \( k \).
• Counting sort is stable! It keeps the records in their original order if they are equal.
Radix Sort

• Intuition:
  – How will you sort YYY-MM-DD?
  – We can sort DD first, then MM, then YYYY?
  – Exactly the same idea with radix sort!

• Sort each digit separately, and starts with the least-significant digit.
  – Actually you can start with the most-significant when input are strings (lexicographic order).

• Assumption:
  – Radix sort must use stable sort for each digit!
Algorithm

\begin{algorithm}
\textbf{RADIUS-SORT}(A, d)
1 \hspace{1em} \textbf{for} i = 1 \textbf{ to } d \\
2 \hspace{1em} \text{use a stable sort to sort array } A \text{ on digit } i
\end{algorithm}

- Normally we can use counting sort with complexity $\Theta(n + k)$.
- The total complexity will be $\Theta(d(n + k))$.
- When $d$ is constant and $k = O(n)$, the complexity will be $\Theta(n)$. 
For any base...

• Base 10:
  − \(1024 = 4 \times 10^0 + 2 \times 10^1 + 0 \times 10^2 + 1 \times 10^3\)
  − Simply compare each digit separately: 4, then 2, then 0, then 1

• Base \(b\):
  − \(x = x_0 \times b^0 + x_1 \times b^1 + \ldots + x_d \times b^d\)
Lemma 8.4
Given \( n \) \( b \)-bit numbers and any positive integer \( r \leq b \), RADIX-SORT correctly sorts these numbers in \( \Theta((b/r)(n + 2^r)) \) time if the stable sort it uses takes \( \Theta(n + k) \) time for inputs in the range 0 to \( k \).

Proof  
For a value \( r \leq b \), we view each key as having \( d = \lfloor b/r \rfloor \) digits of \( r \) bits each. Each digit is an integer in the range 0 to \( 2^r - 1 \), so that we can use counting sort with \( k = 2^r - 1 \). (For example, we can view a 32-bit word as having four 8-bit digits, so that \( b = 32, r = 8, k = 2^r - 1 = 255 \), and \( d = b/r = 4 \).) Each pass of counting sort takes time \( \Theta(n + k) = \Theta(n + 2^r) \) and there are \( d \) passes, for a total running time of \( \Theta(d(n + 2^r)) = \Theta((b/r)(n + 2^r)) \).
Bucket Sort

- **Intuition:**
  - If we can distribute \( n \) numbers into buckets, then sort numbers in each bucket, then just go through all buckets in order and result in the sorted array.

- **Assumption:**
  - Input numbers are uniformly distributed in \([0,1)\).

- **Steps:**
  1. Divide \([0,1)\) into \( n \) equal-sized subintervals (buckets).
  2. Distribute \( n \) numbers into buckets
  3. Expect that each bucket contains few numbers (guaranteed by uniform assumption!)
  4. Sort numbers in each bucket (insertion sort as default).
  5. Then go through buckets in order, listing elements to get sorted array.
Algorithm

**BUCKET-SORT**(A)

1. let \( B[0 \ldots n - 1] \) be a new array
2. \( n = A.length \)
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 \( B[0], B[1], \ldots, B[n - 1] \) together in order
Time Complexity

\[
\begin{aligned}
&\text{BUCKET-SORT}(A) \\
1 & \text{let } B[0..n-1] \text{ be a new array} \\
2 & n = A\. \text{length} \\
3 & \text{for } i = 0 \text{ to } n-1 \\
4 & \quad \text{make } B[i] \text{ an empty list} \\
5 & \text{for } i = 1 \text{ to } n \\
6 & \quad \text{insert } A[i] \text{ into list } B[[i]A[i]] \\
7 & \text{for } i = 0 \text{ to } n-1 \\
8 & \quad \text{sort list } B[i] \text{ with insertion sort} \\
9 & \text{concatenate the lists } B[0], B[1], \ldots, B[n-1] \text{ together in order}
\end{aligned}
\]
Time Complexity

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

Expectation over the input distribution:

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

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

\[
= \Theta(n) + \sum_{i=0}^{n-1} O \left( E \left[ n_i^2 \right] \right) \quad \text{(by equation (C.22))}
\]
Define indicator:

\[ X_{ij} = I\{A[j] \text{ falls in bucket } i\} \]

for \( i = 0, 1, \ldots, n - 1 \) and \( j = 1, 2, \ldots, n \). Thus,

\[ n_i = \sum_{j=1}^{n} X_{ij}. \]

\[ E[n_i^2] = E \left[ \left( \sum_{j=1}^{n} X_{ij} \right)^2 \right] \]

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

\[ = E \left[ \sum_{j=1}^{n} X_{ij}^2 + \sum_{1 \leq j \leq n} \sum_{1 \leq k \leq n, k \neq j} X_{ij} X_{ik} \right] \]

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

\[ = \frac{1}{n^2} + \frac{1}{n}. \]
Time Complexity (Cont’d)

\[ E[n_i^2] = \sum_{j=1}^{n} \frac{1}{n} + \sum_{1 \leq j \leq n} \sum_{1 \leq k \leq n, k \neq j} \frac{1}{n^2} \]

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

\[ = 1 + \frac{n-1}{n} \]

\[ = 2 - \frac{1}{n} , \]

\[ T(n) = \Theta(n) + \sum_{i=0}^{n-1} O(n_i^2) = \Theta(n) + n \times \left(2 - \frac{1}{n}\right) = \Theta(n) \]
Example
Thanks!