Lower Bounds for Comparison-Based Sorting Algorithms (Ch. 8)

We have seen several sorting algorithms that run in $\Omega(n \log n)$ time in the worst case (meaning there is some input on which the algorithm execution takes at least $\Omega(n \log n)$ time).
- merge sort
- heapsort
- quicksort

In all comparison-based sorting algorithms, the sorted order results only from comparisons between input elements.

Is it possible for any comparison-based sorting algorithm to do better?

Lower Bounds for Sorting Algorithms

Fact 1: It takes $\Omega(n)$ time to examine all the input.

Fact 2: All sorting algorithms we've seen so far are $\Omega(n \log n)$ (lower bound)

The Decision Tree Model

Given any comparison-based sorting algorithm, we can represent its behavior on an input of size $n$ by a decision tree. Note: we need only consider the comparisons in the algorithm (the other operations only make the algorithm take longer and we are showing a lower bound).

A decision tree is an abstraction of any comparison-based sorting algorithm.

The Decision Tree Model

A decision tree models all possible execution traces.

- Oval internal nodes correspond to the comparisons in the algorithm.
- Edges are marked with the comparison from a parent to a child node.
- Each rectangular leaf represents one possible ordering of the input for each of $2^n$ possible orderings.
The Decision Tree Model

Example: Insertion sort with n = 3 (3! = 6 leaves)


Note: The length of the longest root to leaf path in this tree = worst-case number of comparisons ≤ worst-case number of operations of algorithm

The Ω(n lg n) Lower Bound

**Lemma 1:** Any binary tree of height h has ≤ 2^h leaves.

**Proof:** By induction on h.

**Basis:** h = 0. Tree is just one node, which is a leaf. 2^h = 2^0 = 1.

**IHOP:** Assume Lemma is true for height = h-1. Extend tree of height h-1 by making as many new leaves as possible. Each leaf in tree of height h-1 becomes the parent of two new leaves.

#leaves for height h = 2(# leaves for height h-1) = 2(2^{h-1}) (by the IHOP) = 2^h QED

The Ω(n lg n) Lower Bound

**Theorem:** Any decision tree T for sorting n elements has height Ω(n lg n) (therefore, any comparison sort algorithm requires Ω(n lg n) comparisons in the worst case).

**Proof:** Let h be the height of T. Then we know:
- T has at least n! leaves (all permutations must be in tree).
- T is binary, so for height h, it has at most 2^h leaves (L.1)

2^h ≥ #leaves ≥ n!

2^h ≥ n!

lg(2^h) ≥ lg(n!) * Take lg of both sides *

h ≥ Ω(n lg n) (Stirling's Approx) □

Therefore, heapsort and mergesort are asymptotically optimal comparison sorts.

Beating the lower bound ... non-comparison sorts

**Idea:** Algorithms that are NOT comparison-based might be faster.

There are three such algorithms presented in Chapter 8:
- counting sort
- radix sort
- bucket sort

These algorithms:
- run in O(n) time (if certain conditions can be guaranteed)
- either use information about the values to be sorted (counting sort, bucket sort), or
- operate on "pieces" of the input elements (radix sort)

Counting Sort

Counting sort is an algorithm for sorting a collection of objects according to keys that are small integers; that is, it is an integer sorting algorithm.

It operates by counting the number of objects that have each distinct key value, and uses arithmetic on those counts to determine the positions of each key value in the output sequence.

Its running time is linear in the number of items and the maximum key value, so it is only suitable for use in situations where the keys are not significantly larger than the number of items. It is often used as a subroutine in another sorting algorithm, radix sort.
Counting Sort

**Key Assumption:** input elements are integers in known range \([0..k]\) for some constant \(k\).

**Idea:** for each input element \(x\), find the number of elements \(< x\) (say this number = \(m\)) and put \(x\) in the \((m+1)st\) spot in the output array.

Counting-Sort(A, k)
// \(A[1..n]\) is input array, \(C[0..k]\) is initially all 0's, \(B[1..n]\) is output array
1. for \(i = 1\) to A.length
2. \(C[A[i]] = C[A[i]] + 1\) // Each element of \(C\) is a counter for the value at \(A[i]\) (if \(i\) holds number of 6s in \(A\)).
3. for \(i = 1\) to \(k\)
4. \(C[i] = C[i] + C[i-1]\) // Make \(C\) into a "prefix sum" array, where \(C[i]\) contains number of elements \(\leq i\)
5. for \(j = A.length\ downto 1\)
7. \(C[A[j]] = C[A[j]] - 1\)

See in-class example (another example in textbook).

Running Time of Counting Sort

for loop in lines 1-2 takes \(\theta(mn)\) time. Overall time is \(\theta(k + n)\).
for loop in lines 3-4 takes \(\theta(k)\) time.
for loop in lines 5-7 takes \(\theta(n)\) time.

In practice, use counting sort when we have \(k = \Theta(n)\), so running time is \(\Theta(n)\).

Counting sort has the important property of **stability**:
A sorting algorithm is **stable** when numbers with the same values appear in the output array in the same order as they do in the input array.

Stability is important when satellite data is stored with elements being sorted and when counting sort is used as a subroutine for radix sort, an algorithm that relies on a stable sorting subroutine.

Radix Sort

How IBM made its money! Punch cards readers were used for census tabulation in early 1900s. Card sorters worked on one column at a time, human operators re-entered the cards to sort on the next column. **The human operator was part of the algorithm!**

Let \(d\) be the number of digits in each input number.

Let \(d\) be the number of digits in each input number.

Radix-Sort(A, d)
1. for \(i = 1\) to \(d\) do
2. use stable sort to sort array \(A\) on digit \(i\)

Note:
- radix sort sorts the least significant digit first.
- correctness can be shown by induction on the digit being sorted.
- often counting sort is used as the internal sort in step 2.
**Running Time of Radix Sort**

Let $d$ be the number of digits in each input number.

```
Radix-Sort(A, d)
1. for $i$ % 1 to $d$ do
2. use stable sort to sort array A on digit i
```

Running time of radix sort: $O(dT_{ss}(n))$
- $T_{ss}$ is the time for the internal sort. Counting sort gives $T_{ss}(n) = O(k + n)$, so $O(dT_{ss}(n)) = O(d(k + n))$.
- If $d = O(\lg n)$ and $k = 2$ (common for computers), then $O(d(k + n)) = O(n\lg n)$.

**Correctness of Radix Sort**

Use induction on the number of passes $i$.

Loop Invariant: Before iteration $i$ of the Radix Sort algorithm, digits 1...$i-1$ are sorted.

Basis: Prior to the first pass, digits 1...0 are trivially sorted.

Maintenance: (IHOP) Assume digits 1...$i-1$ are sorted before ith pass.

Inductive step: Show that a stable sort on digit $i$ leaves digits 1...$i$ sorted.
- If 2 digits in position $i$ are different, ordering by position $i$ is correct and positions 1...$i-1$ are irrelevant.
- If 2 digits in position $i$ are equal, numbers are already in right order (by IHOP). The stable sort on digit $i$ leaves them in the right order.

**Bucket Sort**

1. Divide $[0,1)$ into $n$ equal-sized buckets.
2. Distributed the $n$ values into the buckets.
3. Sort each bucket.
4. Then go through buckets in order, listing elements in each one

Input: $A[1...n]$, where $0 \leq A[i] < 1$ for all $i$, and $x$, a real number

Auxilliary array: $B[0...n-1]$ of linked lists, each list initially empty.

```
Bucket-Sort(A, x)
1. n = length[A]
2. for $i$ = 1 to n
3. insert A[$i$] into list $B[\text{floor of } xA[i]]$
4. for $i$ = 0 to n-1
5. sort list $i$ with Insertion-Sort
6. Concatenate lists $B[0], B[1],...,B[n-1]$
```

**Bucket Sort**

Assumption: input elements distribute uniformly over some known range, e.g., $[0,1)$, so all elements in A are greater than or equal to 0 but less than 1. Line 3 multiplies the element in A by a fraction small enough to produce a number in $[0,1)$.

```
Bucket-Sort(A, x)
1. n = length[A]
2. for $i$ = 1 to n
3. insert A[$i$] into list $B[\text{floor of } xA[i]]$
4. for $i$ = 0 to n-1
5. sort list $i$ with Insertion-Sort
6. Concatenate lists $B[0], B[1],...,B[n-1]$
```

A = [78, 17, 39, 26, 72, 94, 21, 12, 23, 68] Multiply by $x = 0.01$
**Bucket Sort**

Bucket-Sort(A, x)
1. \( n = \text{length}[A] \)
2. for \( i = 1 \) to \( n \)
3. insert \( A[i] \) into list \( B[\text{floor of } xA[i]] \)
4. for \( i = 0 \) to \( n-1 \)
5. sort list \( i \) with Insertion-Sort
6. Concatenate lists \( B[0], B[1], \ldots, B[n-1] \)

Running time of bucket sort: \( O(n) \) expected time if URD of data
Lines 2-3: \( O(1) \) for each interval = \( O(n) \) time total.
Lines 4-5: \( O(n) \) to \( O(n^2) \)
Line 6: \( O(n) \) time to scan the \( n \) buckets containing a total of \( n \) input elements

**Summary NCB Sorts**

Non-Comparison-Based Sorts

<table>
<thead>
<tr>
<th>Sort</th>
<th>Running Time</th>
<th>worst-case</th>
<th>average-case</th>
<th>best-case</th>
<th>in place</th>
</tr>
</thead>
<tbody>
<tr>
<td>Counting Sort</td>
<td>( O(n + k) )</td>
<td>( O(n + k) )</td>
<td>( O(n + k) )</td>
<td>no</td>
<td></td>
</tr>
<tr>
<td>Radix Sort</td>
<td>( O(n + k') )</td>
<td>( O(n + k') )</td>
<td>( O(n + k') )</td>
<td>no</td>
<td></td>
</tr>
<tr>
<td>Bucket Sort</td>
<td>( O(n) )</td>
<td></td>
<td></td>
<td>no</td>
<td></td>
</tr>
</tbody>
</table>

How do these algorithms beat the lower bound for sorting?

- **Counting sort** assumes input elements are in range \([0,1,2,..,k]\) and uses array indexing to count the number of occurrences of each value.
- **Radix sort** assumes each integer consists of \( d \) digits, and each digit is in range \([1,2,..,k']\).
- **Bucket sort** requires advance knowledge of input distribution or randomization technique (sorts \( n \) numbers uniformly distributed in range in \( O(n) \) time).