Sorting: Lower Bounds and Linear Time

Last Revision: August 25, 2016

Lower Bound for Sorting

- All sorting algorithms we seen so far are based on comparing elements
  - E.g., Insertion sort, Selection sort, Mergesort, Heapsort and Quicksort
- Insertion sort, Selection sort and Quicksort have worst-case running times $\Theta(n^2)$, while the others have worst-case running time $\Theta(n \log n)$

Question

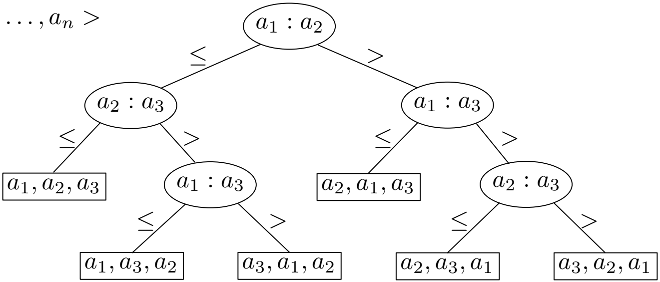
Can we do better?

Goal

We will prove that any comparison-based sorting algorithm has a worst-case running time $\Omega(n \log n)$.

Decision-tree Example

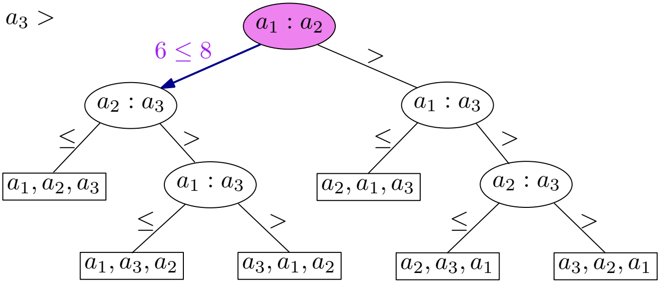
Sort $< a_1, a_2, \ldots, a_n >$



- Each internal node is labeled $a_i : a_j$ for $\{1, 2, \ldots, n\}$
  - The left subtree shows subsequent comparisons if $a_i \leq a_j$
  - The right subtree shows subsequent comparisons if $a_i > a_j$
- Each leaf corresponds to an input ordering

Decision-tree Example

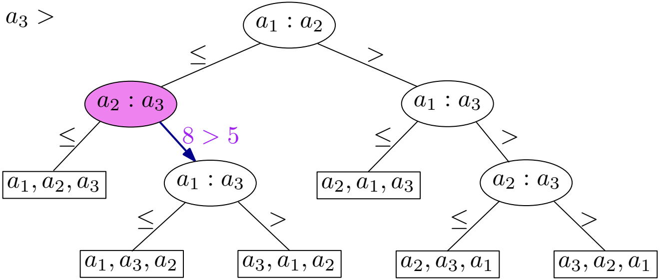
Sort $< a_1, a_2, a_3 >$
$=< 6, 8, 5 >$:



- Each internal node is labeled $a_i : a_j$ for $\{1, 2, \ldots, n\}$
  - The left subtree shows subsequent comparisons if $a_i \leq a_j$
  - The right subtree shows subsequent comparisons if $a_i > a_j$
- Each leaf corresponds to an input ordering

Decision-tree Example

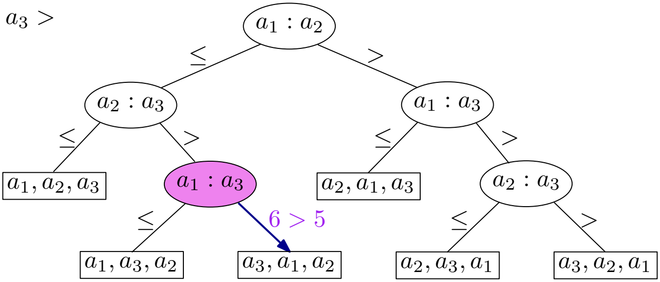
Sort $< a_1, a_2, a_3 >$
$=< 6, 8, 5 >$:



- Each internal node is labeled $a_i : a_j$ for $\{1, 2, \ldots, n\}$
  - The left subtree shows subsequent comparisons if $a_i \leq a_j$
  - The right subtree shows subsequent comparisons if $a_i > a_j$
- Each leaf corresponds to an input ordering

Decision-tree Example

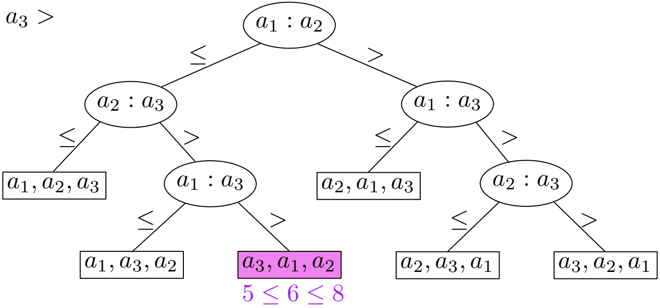
Sort $< a_1, a_2, a_3 >$
$=< 6, 8, 5 >$:



- Each internal node is labeled $a_i : a_j$ for $\{1, 2, \ldots, n\}$
  - The left subtree shows subsequent comparisons if $a_i \leq a_j$
  - The right subtree shows subsequent comparisons if $a_i > a_j$
- Each leaf corresponds to an input ordering

Decision-tree Example

Sort $< a_1, a_2, a_3 >$
$=< 6, 8, 5 >$:



- Each internal node is labeled $a_i : a_j$ for $\{1, 2, \ldots, n\}$
  - The left subtree shows subsequent comparisons if $a_i \leq a_j$
  - The right subtree shows subsequent comparisons if $a_i > a_j$
- Each leaf corresponds to an input ordering

Decision-tree Model

A decision tree can model the execution of **any** comparison-based sorting algorithm

- One tree for each input size n
- Worst-case running time = height of tree

Lower Bound for Sorting

Theorem

Any comparison-based sorting algorithm requires $\Omega(n \log n)$ comparisons in the worst case.

Proof.

- A decision tree to sort n elements must have at least $n!$ leaves, since each of the $n!$ orderings is a possible answer.
- A binary tree of height h has at most 2^h leaves
- Thus, $n! \leq 2^h$
  $\Rightarrow h \geq \log n! = \Omega(n \log n)$ (Stirling's approximation)

$\square$

Corollary

Heapsort and merge sort are asymptotically optimal comparison-based sorting algorithms.

Lower Bound for Average Running Time of Sorting

- We just proved that worst case number of comparisons used is $\Omega(n \log n)$
- Suppose that each of the $n!$ input permutations is equally likely. What can be said about the average case running time?

Note: Average is taken by adding up individual running time of algorithm on each possible input and dividing total by $n!$.

Theorem

When all input permutations are equally likely, any comparison-based sorting algorithm requires $\Omega(n \log n)$ comparisons on average.

Lower Bound for Average Running Time of Sorting

Theorem

When all input permutations are equally likely, any comparison-based sorting algorithm requires $\Omega(n \log n)$ comparisons on average.

Proof.

- The *External Path Length (EPL)* of a tree is the sum over all leaves of the tree, of the length of the paths from the root to the leaves.
- Average number of comparisons used by a sorting algorithm is EPL of its associated comparison tree divided by $n!$.
- The EPL of a binary tree with m leaves is at least $m \log_2 m + O(m)$.
- The comparison tree has $m = n!$ leaves
  $\Rightarrow$ its external path length is $n! \log_2 n! + O(n!)$
  $\Rightarrow$ average number of comparisons used is $\log_2 n! + O(1)$.
- We already saw $\log_2 n! = \Omega(n \log n)$.

$\square$

Can we do better?

Are there sorting algorithms which are not comparison-based?
Can they beat the $\Omega(n \log n)$ lower bound?

- Counting sort
  - Assumes items are in set $\{1, 2, \ldots, k\}$.
  - Is a *stable* sort (defined soon).
- Radix sort
  - Assumes items are stored in fixed size words using finite alphabet

Counting-sort(A, B, k)

```
Input: A[1...n], where A[j] ∈ {1, 2, ..., k}
Output: B[1...n], sorted
let C[1...k] be a new array;
for i ← 1 to k do
  | C[i] ← 0;
end
for j ← 1 to n do
  | C[A[j]] ← C[A[j]] + 1; // C[i] = |{key = i}|
end
for i ← 2 to k do
  | C[i] ← C[i] + C[i − 1]; // C[i] = |{key ≤ i}|
end
for j ← n to 1 do
  | B[C[A[j]]] ← A[j];
  | C[A[j]] ← C[A[j]] − 1;
end
```

Example: Counting Sort

	1	2	3	4	5
A	4	2	1	4	2

	1	2	3	4
C				

B					

Example: Counting Sort

	1	2	3	4	5
A	4	2	1	4	2

	1	2	3	4
C	0	0	0	0

B					

for $i \leftarrow 1$ **to** k **do**
  $C[i] \leftarrow 0;$
end

Example: Counting Sort

	1	2	3	4	5
A	4	2	1	4	2

	1	2	3	4
C	0	0	0	1

B					

for $j \leftarrow 1$ **to** n **do**

$\quad C[A[j]] \leftarrow C[A[j]] + 1$; // $C[i] = |\{\text{key} = i\}|$

end

Example: Counting Sort

	1	2	3	4	5
A	4	2	1	4	2

	1	2	3	4
C	0	1	0	1

B					

for $j \leftarrow 1$ **to** n **do**

  $C[A[j]] \leftarrow C[A[j]] + 1$; // $C[i] = |\{\text{key} = i\}|$

end

Example: Counting Sort

	1	2	3	4	5
A	4	2	1	4	2

	1	2	3	4
C	1	1	0	1

B					

for $j \leftarrow 1$ **to** n **do**
  $C[A[j]] \leftarrow C[A[j]] + 1$; // $C[i] = |\{\text{key} = i\}|$
end

Example: Counting Sort

	1	2	3	4	5
A	4	2	1	4	2

	1	2	3	4
C	1	1	0	2

B					

for $j \leftarrow 1$ **to** n **do**
   $C[A[j]] \leftarrow C[A[j]] + 1$; // $C[i] = |\{\text{key} = i\}|$
end

A

1	2	3	4	5
4	2	1	4	2

C

1	2	3	4
1	2	0	2

B

for $j \leftarrow 1$ **to** n **do**

  $C[A[j]] \leftarrow C[A[j]] + 1$; // $C[i] = |\{\text{key} = i\}|$

end

Example: Counting Sort

	1	2	3	4	5
A	4	2	1	4	2

	1	2	3	4
C	1	2	0	2

B					

C'	1	3	0	2

for $i \leftarrow 2$ **to** k **do**
  $C[i] \leftarrow C[i] + C[i-1]$; // $C[i] = |\{\text{key} \leq i\}|$
end

Example: Counting Sort

	1	2	3	4	5
A	4	2	1	4	2

	1	2	3	4
C	1	2	0	2

B					

C'	1	3	3	2

for $i \leftarrow 2$ **to** k **do**

$\quad C[i] \leftarrow C[i] + C[i-1];$ // $C[i] = |\{\text{key} \leq i\}|$

end

Example: Counting Sort

A

1	2	3	4	5
4	2	1	4	2

C

1	2	3	4
1	2	0	2

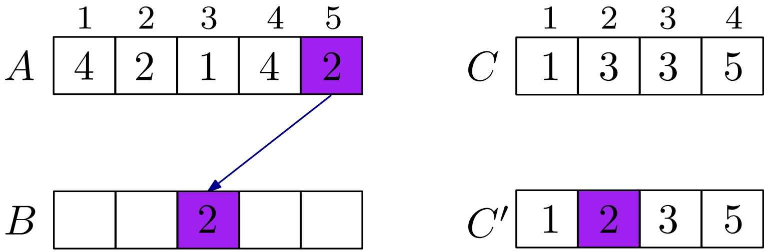
B

C'

1	3	3	5

for $i \leftarrow 2$ **to** k **do**
  $C[i] \leftarrow C[i] + C[i-1]$; // $C[i] = |\{\text{key} \leq i\}|$
end

Example: Counting Sort



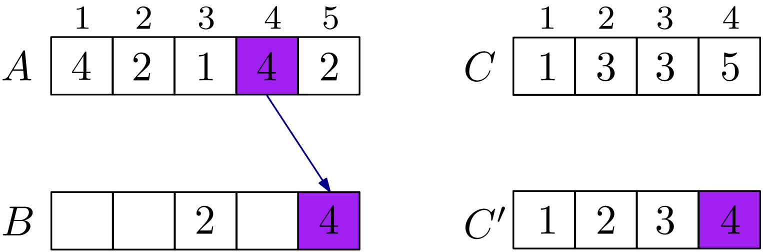
for $j \leftarrow n$ **to** 1 **do**

$\quad B[C[A[j]]] \leftarrow A[j];$

$\quad C[A[j]] \leftarrow C[A[j]] - 1;$

end

Example: Counting Sort



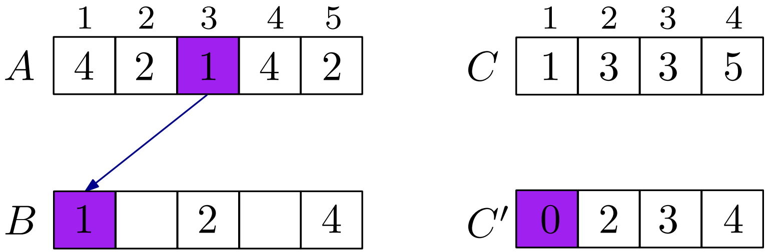
for $j \leftarrow n$ **to** 1 **do**

$\quad B[C[A[j]]] \leftarrow A[j];$

$\quad C[A[j]] \leftarrow C[A[j]] - 1;$

end

Example: Counting Sort



for $j \leftarrow n$ **to** 1 **do**

$\quad B[C[A[j]]] \leftarrow A[j];$

$\quad C[A[j]] \leftarrow C[A[j]] - 1;$

end

	1	2	3	4	5
A	4	2	1	4	2
B	1	2	2		4

	1	2	3	4
C	1	3	3	5
C'	0	1	3	4

for $j \leftarrow n$ **to** 1 **do**

$B[C[A[j]]] \leftarrow A[j]$;

$C[A[j]] \leftarrow C[A[j]] - 1$;

end

	1	2	3	4	5
A	4	2	1	4	2
B	1	2	2	4	4

	1	2	3	4
C	1	3	3	5
C'	0	1	3	3

$$
\begin{array}{l}
\textbf{for } j \leftarrow n \textbf{ to } 1 \textbf{ do} \\
\quad B[C[A[j]]] \leftarrow A[j]; \\
\quad C[A[j]] \leftarrow C[A[j]] - 1; \\
\textbf{end}
\end{array}
$$

Analysis

Input: $A[1 \dots n]$, where $A[j] \in \{1, 2, \dots, k\}$
Output: $B[1 \dots n]$, sorted
let $C[1 \dots k]$ be a new array;
for $i \leftarrow 1$ **to** k **do**
    $C[i] \leftarrow 0$; // $O(k)$
end
for $j \leftarrow 1$ **to** n **do**
    $C[A[j]] \leftarrow C[A[j]] + 1$; // $O(n)$
end
for $i \leftarrow 2$ **to** k **do**
    $C[i] \leftarrow C[i] + C[i-1]$; // $O(k)$
end
for $j \leftarrow n$ **to** 1 **do**
    $B[C[A[j]]] \leftarrow A[j]$;
    $C[A[j]] \leftarrow C[A[j]] - 1$; // $O(n)$
end

Total: $O(n+k)$

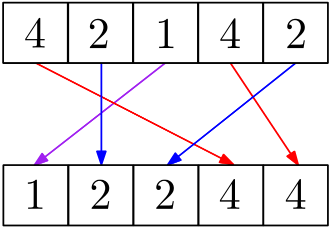
Running Time

If $k = O(n)$, then counting sort takes $O(n)$ time.

- But didn't we prove that sorting must take $\Omega(n \log n)$ time?
- No, actually we proved that any comparison-based sorting algorithm takes $\Omega(n \log n)$ time.
- Note that counting sort is *not* a comparison-based sorting algorithm.
- In fact, it makes no comparisons at all!

Counting sort is a stable sort

- it preserves the input order among equal elements.



Exercise

What other sorts have this property?

Radix Sort

- Sort on *least significant* digit first using stable sort

2 3 2 9	2 7 2 0	2 7 2 0	2 3 2 9	2 3 2 9
5 4 5 7	5 3 5 5	2 3 2 9	5 3 5 5	2 7 2 0
3 6 5 7	3 4 3 6	3 4 3 6	3 4 3 6	3 4 3 6
5 8 3 9	5 4 5 7	5 8 3 9	5 4 5 7	3 6 5 7
3 4 3 6	3 6 5 7	5 3 5 5	3 6 5 7	5 3 5 5
2 7 2 0	2 3 2 9	5 4 5 7	2 7 2 0	5 4 5 7
5 3 5 5	5 8 3 9	3 6 5 7	5 8 3 9	5 8 3 9

Radix Sort: Correctness

Induction on digit position

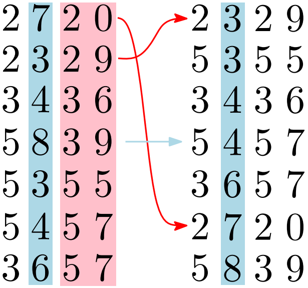
- Assume that the numbers are sorted by their low-order $i-1$ digits
- Sort on digit i

2	7	2	0		2	3	2	9
2	3	2	9		5	3	5	5
3	4	3	6		3	4	3	6
5	8	3	9	$\longrightarrow$	5	4	5	7
5	3	5	5		3	6	5	7
5	4	5	7		2	7	2	0
3	6	5	7		5	8	3	9

Induction on digit position

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

- Sort on digit i
  - Two numbers that differ on digit i are correctly sorted by their low-order i digits



Induction on digit position

- Assume that the numbers are sorted by their low-order $i-1$ digits
- Sort on digit i
  - Two numbers that differ on digit i are correctly sorted by their low-order i digits
  - Two numbers equal on digit i are put in the same order as the input $\Rightarrow$ correctly sorted by their low-order i digits

2 7 2 0	2 3 2 9
2 3 2 9	5 3 5 5
3 4 3 6	3 4 3 6
5 8 3 9	5 4 5 7
5 3 5 5	3 6 5 7
5 4 5 7	2 7 2 0
3 6 5 7	5 8 3 9

Radix Sort: Running Time & Application

Lemma

Given n d-digit numbers in which each digit can take on up to k possible values, radix sort correctly sorts these numbers in $O(d(n+k))$ time if the stable sort it uses takes $O(n+k)$ time.

Application:
Sorting numbers in the range from 0 to $n^b - 1$, where b is a constant

- $b \log n$ bits for each number
- each number can be viewed as having $O(b)$ digits of $\log n$ bits each
- running time is $O(d(n+k)) = O(b(n+2^{\log n})) = O(bn)$
- since b is a constant, the running time is $O(n)$.