Lower Bounds for Comparison Based Sorting
and
Sorting Digital Data
Lower bounds for comparison-based sorting

Consider only *deterministic* algorithms that make comparisons “\(a_i \leq a_j\)"

**Formally:** model such an algorithm as a *decision tree*:

- each internal node labeled by a pair of indices \((i, j)\), meaning compare \(a_i\) with \(a_j\)
  - two children: left branch taken if \(a_i \leq a_j\), right branch taken if \(a_i > a_j\)
- each leaf is labeled by a permutation on \(\{1, \ldots, n\}\), indicating the sorted order
- Cost = height of tree (max # of comparisons made on any input)
Example: Merge Sort on \( n = 3 \)
**Theorem.** Any decision tree that correctly sorts \( n \) items must have height at least \( n \log_2(n) - n/\ln(2) \)

_This means:_ for every deterministic sorting algorithm \( A \) and every \( n \), there exists an input \( L \) of size \( n \) on which \( A \) makes at least \( n \log_2(n) - n/\ln(2) \) comparisons

**Proof.** All \( n! \) permutations must appear as leaves

Fact: there are at most \( 2^j \) leaves at depth at most \( j \)

\[ \therefore \text{if } h = \text{height of tree:} \]

\[ 2^h \geq n! \implies h \geq \log_2 n! \]

**Claim.** \( \log_2 n! \geq n \log_2 n - n/\ln(2) \)

Recall: Approximating sums by integrals

If \( f \) is continuous and monotone on \([a, b]\), \( m := \min(f(a), f(b))\), and \( M := \max(f(a), f(b))\):

\[
\int_{a}^{b} f(x)dx + m \leq \sum_{i=a}^{b} f(i) \leq \int_{a}^{b} f(x)dx + M
\]
Proof of claim

We have

\[ \log_2 n! = \sum_{i=1}^{n} \log_2 i \]

and

\[ \int \ln(x) dx = x(\ln(x) - 1) \]

due to the change of variable \( u = \ln(x) \),

therefore

\[ \log_2 n! \geq \int_{1}^{n} \log_2 x dx \geq n \log_2 n - n/\ln(2) \]
The same argument shows that the algorithm must make at least $\frac{1}{2} n \log_2 n$ comparisons for the vast majority of inputs:

$$\Pr[ \text{# comparisons} \leq \frac{1}{2} n \log_2 n] \leq 2^{\left(-\frac{1}{2} + o(1)\right)n \log_2 n}$$

(Recall: $f(n) = o(1)$ means $f(n) \to 0$ as $n \to \infty$)

This holds for every deterministic algorithm, and a randomly permuted input.

This follows from the fact that there are at most

$$2^{\frac{1}{2} n \log_2 n}$$

leaves at depth $\leq \frac{1}{2} n \log_2 n$, and

$$n! \geq 2^{(1+o(1))n \log_2 n}$$

leaves in total.
Extension: probabilistic algorithms

We can view a probabilistic sorting algorithm $A$ as a deterministic algorithm that $B$ takes two inputs: a list $L$ to sort, and string $R$ representing the random choices made by $A$.

To run $A$ on input $L$, choose $R$ at random and run $B$ on inputs $L$ and $R$.

The above analysis shows that for every $R$, at least half of the $L$’s make $B$ run for time at least $t := \frac{1}{2} n \log_2 n$ steps.

It follows that for at least one $L$, at least half of the $R$’s make $B$ run for at least $t$ steps — why?

Imagine an array with rows indexed by $R$ and columns indexed by $L$.

Color each cell black if $B(L, R)$ takes at least $t$ steps, o/w color it white.

We know: in each row, the density of black cells is at least $\frac{1}{2}$.

Thus, the overall density of black cells in the array is at least $\frac{1}{2}$.

∴ at least one column must have density at least $\frac{1}{2}$.

So for at least one input $L$, the expected number of comparisons made by $A$ is at least $t/2$. 


Bucket Sort

Let \( \Delta = \{0, \ldots, R - 1\} \)

- **input:** \( a_1, \ldots, a_n \in \Delta \)
- for \( k \) in \([0..R)\) do: \( B[k] \leftarrow \text{emptyList} \)
- for \( i \) in \([1..n]\) do
  - append \( a_i \) to \( B[a_i] \)
- return \( \text{concat}(B[0], B[1], \ldots, B[R - 1]) \)

Running time: \( O(n + R) \)

Note:
- this is a “stable” sort
LSD Radix Sort

An algorithm to sort strings

Processes characters from least significant to most

*input:* a list $L$ of $n$ strings over $\Delta$, each of length $\ell$

for $j$ in reverse $[1..\ell]$ do
  bucket sort $L$ using the $j$th entry of each string as the key

Correctness: follows from stability of Bucket Sort

Running time: $O(n\ell + R\ell)$
LSD Radix sort in practice:

IBM Type 82 Sorter (1949)

IBM Card Sorters deal cards from a source deck into 13 output pockets (one pocket for rejects plus one pocket for each of the 12 rows on the card) at the rate of 250-2000 cards per minute, depending on model. One column, selected by the Sorting Brush, is sorted per pass. The Selector Switches determine which row(s) in each column are included in the sort. To fully sort a deck of cards required a number of passes through the sorter; one pass per column in the sort key. Sorters normally did not have a control panel (plugboard), but Type 75, 80, and 82 sorters had a tiny one if they were equipped with the optional Multiple Column Selection Device.
If $R \leq n$, running time is $O(n\ell)$, which is linear in the input size $n\ell$

- Lower bound says we need $\approx n \log_2 n$ string comparisons, and each string comparison can take time $\approx \ell$
- So we should have a lower bound of $\approx n \log_2 n \cdot \ell$
- Contradiction? No!

Improvements:

- handle variable length inputs
- running time: to $O(N + R\ell)$, where $N := $ is the sum of the lengths of all input strings, and $\ell := $ max length of input strings
- *key indexed counting*: a more efficient alternative to using linked lists (below)
LSD Radix Sort: variable length inputs

**input:** a list $L$ of $n$ strings over $\Delta$, each of length at most $\ell$

bucket sort $L$ into an array of lists $B[0..\ell]$ such that $B[j]$ contains all $s$ in $L$ with $\text{length}(s) = j$

$L \leftarrow \text{emptyList}$

for $j$ in reverse $[1..\ell]$ do

$L \leftarrow \text{concat}(B[j], L)$

bucket sort $L$ using the $j$th entry of each string as the key

return $\text{concat}(B[0], L)$
Key indexed counting

Idea: avoid using linked lists

Assume we are sorting an array $A[0..n)$, where each $A[i]$ is a pointer to an array of $\ell$ characters

We need an auxiliary array $\text{temp}[0..n)$ of array pointers, and array $\text{count}[0..R)$ of integers

for $j$ in reverse $[0..\ell)$

// compute frequency count of each character in position $j$
for $k$ in $[0..R]$ do $\text{count}[k] \leftarrow 0$
for $i$ in $[0..n)$ do $\text{count}[A[i][j] + 1]++$

// compute cumulative counts, so that character $k$’s “bucket” is $\text{temp}[\text{count}[k]..\text{count}[k + 1])$
for $k$ in $[0..R)$ do $\text{count}[k + 1] += \text{count}[k]$

// copy each string to the right bucket
for $i$ in $[0..n)$ do $\text{temp}[\text{count}[A[i][j]]++]) \leftarrow A[i]$

// copy strings back to $A$
for $i$ in $[0..n)$ do $A[i] \leftarrow \text{temp}[i]$
MSD Radix Sort

Processes characters from most significant to least

Easy to implement to work with variable length strings

Can run in \textit{sub-linear} time, since it does not necessarily inspect all characters (same holds for comparison based sorts)

Basic idea:

Sort strings into buckets based on first character of each string

Recursively sort each bucket (based on the suffixes of each string)

Gather together all of the sorted buckets
More details ...

Input: a list $L$ of length $n$ of variable length strings over $\Delta = \{0, \ldots, R - 1\}$

For simplicity, assume the 0 character acts as a string terminator

$$MSDRadixSort(L, j): \text{// sort } L \text{ based on positions } j, j+1, \ldots$$

if length($L$) is very small then
    sort $L$ using another algorithm and return

bucket sort $L$ into an array of lists $B[0..R)$ such that $B[k]$ contains all $s$ in $L$ with $s[j] = k$

for $k$ in $[1..R)$ do: $MSDRadixSort(B[k], j + 1)$

return $concat(B[0], B[1], \ldots, B[R-1])$

Initial call: $MSDRadixSort(L, 0)$

Implementation: Use “key indexed counting” to do the bucket sorting
Analysis of MSD Radix Sort

Sorting into buckets takes at least $R$ steps, so it does not make sense to use it on lists of length less than $R$.

The algorithm need not look at all characters: it may be possible to sort strings based on short prefixes.

The distinguishing prefix of $s$ in $L$: the shortest prefix of $s$ that is not a prefix of any other string in $L$ (or $s$ if $s$ is a prefix of another string).

alignment
all
allocate
alphabet
alternate
alternative

Any sorting algorithm needs to look at all characters in the distinguishing prefixes (and no others).

Define $\text{DP}(L) = \sum_{s \in L} \text{(length of distinguishing prefix of } s \text{ in } L)$

Running time of MSD Radix Sort: $O(\text{DP}(L) + n \log R)$

- Assumes small lists are sorted with String Quick Sort (below)
String Quick Sort

A blend of MSD Radix Sort and Quick Sort

\texttt{StringQuickSort}(L, j): // sort \( L \) based on positions \( j, j+1, \ldots \)

\begin{itemize}
  \item if length(\( L \)) is very small then
    sort \( L \) using another algorithm and return
  \item choose \( t \) in \( L \) at random and set \( p \leftarrow t[j] \)
  \item partition \( L \) into 3 sublists: // use fast 3-way partition algorithm
    \begin{itemize}
      \item \( L_{<p}: \) \( s \) in \( L \) with \( s[j] < p \)
      \item \( L_{=p}: \) \( s \) in \( L \) with \( s[j] = p \)
      \item \( L_{>p}: \) \( s \) in \( L \) with \( s[j] > p \)
    \end{itemize}
  \item \texttt{StringQuickSort}(\( L_{<p}, j \))
  \item if \( p \neq 0 \) then \texttt{StringQuickSort}(\( L_{=p}, j + 1 \))
    // don’t recurse when we’re at the end of string
  \item \texttt{StringQuickSort}(\( L_{>p}, j \))
  \item return \texttt{concat}(\( L_{<p}, L_{=p}, L_{>p} \))
\end{itemize}

Initial call: \texttt{StringQuickSort}(\( L, 0 \))
Running time of String Quick Sort:
\( O(DP(L) + n \log n) \)

- The \( O(n \log n) \) term is an expected running time (that holds for arbitrary inputs)

Experimental running times:

- \( n = 10^7, 10^6, 10^5, 10^4 \)
- character strings of variable length at most 15
- distribution mimicking English words

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>( 10^7 ) (s)</th>
<th>( 10^6 ) (s/10)</th>
<th>( 10^5 ) (s/10^2)</th>
<th>( 10^4 ) (s/10^3)</th>
</tr>
</thead>
<tbody>
<tr>
<td>system</td>
<td>1.462*</td>
<td>1.308</td>
<td>1.076</td>
<td>0.871</td>
</tr>
<tr>
<td>qsort</td>
<td>1.387</td>
<td>1.232</td>
<td>1.084</td>
<td>0.919</td>
</tr>
<tr>
<td>merge</td>
<td>1.945</td>
<td>1.641</td>
<td>1.334</td>
<td>0.978</td>
</tr>
<tr>
<td>string qsort</td>
<td>0.728</td>
<td>0.664</td>
<td>0.607</td>
<td>0.541</td>
</tr>
<tr>
<td>MSD radix</td>
<td>0.323</td>
<td>0.276</td>
<td>0.270</td>
<td>0.266</td>
</tr>
<tr>
<td>LSD radix</td>
<td>0.562</td>
<td>0.337</td>
<td>0.331</td>
<td>0.282</td>
</tr>
</tbody>
</table>

* “system” = IntroSort: \( 2 \log_2 n \) levels of QuickSort, follows by HeapSort
Experimental running times:

- \( n = 10^7, 10^6, 10^5, 10^4 \)
- character strings of fixed length 63
- uniform distribution over the 26 letters of the alphabet

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>(10^7) (s)</th>
<th>(10^6) (s/10)</th>
<th>(10^5) (s/10^2)</th>
<th>(10^4) (s/10^3)</th>
</tr>
</thead>
<tbody>
<tr>
<td>system</td>
<td>2.065</td>
<td>1.665</td>
<td>1.336</td>
<td>1.016</td>
</tr>
<tr>
<td>qsort</td>
<td>2.186</td>
<td>1.774</td>
<td>1.430</td>
<td>1.111</td>
</tr>
<tr>
<td>merge</td>
<td>3.273</td>
<td>2.604</td>
<td>1.995</td>
<td>1.295</td>
</tr>
<tr>
<td>string qsort</td>
<td>1.774</td>
<td>1.402</td>
<td>1.150</td>
<td>0.948</td>
</tr>
<tr>
<td>MSD radix</td>
<td>1.277</td>
<td>0.874</td>
<td>0.610</td>
<td>0.598</td>
</tr>
</tbody>
</table>

**Caution:** timings are very sensitive to implementation details

Sort array of `char[KEYLEN]` rather than `char*` to avoid bad cache behavior

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>char[KEYLEN]</th>
<th>char*</th>
</tr>
</thead>
<tbody>
<tr>
<td>system</td>
<td>2.065</td>
<td>6.210</td>
</tr>
<tr>
<td>qsort</td>
<td>2.186</td>
<td>5.570</td>
</tr>
<tr>
<td>merge</td>
<td>3.273</td>
<td>6.286</td>
</tr>
<tr>
<td>string qsort</td>
<td>1.774</td>
<td>3.806</td>
</tr>
<tr>
<td>MSD radix</td>
<td>1.277</td>
<td>5.354</td>
</tr>
<tr>
<td>LSD radix</td>
<td>19.355</td>
<td>117.748</td>
</tr>
</tbody>
</table>
Experimental running times for sorting integers:

• $n = 10^7, 10^6, 10^5, 10^4$
• 32-bit integers
• uniform distribution

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>$10^8$ (s)</th>
<th>$10^7$ (s)</th>
<th>$10^6$ (s/10)</th>
<th>$10^5$ (s/10^2)</th>
<th>$10^4$ (s/10^3)</th>
</tr>
</thead>
<tbody>
<tr>
<td>system</td>
<td>0.806</td>
<td>0.707</td>
<td>0.613</td>
<td>0.507</td>
<td>0.431</td>
</tr>
<tr>
<td>MSD radix</td>
<td>0.233</td>
<td>0.347</td>
<td>0.175</td>
<td>0.309</td>
<td>0.189</td>
</tr>
<tr>
<td>LSD radix</td>
<td>0.349</td>
<td>0.341</td>
<td>0.132</td>
<td>0.132</td>
<td>0.099</td>
</tr>
</tbody>
</table>

• 64-bit integers
• uniform distribution

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>$10^8$ (s)</th>
<th>$10^7$ (s)</th>
<th>$10^6$ (s/10)</th>
<th>$10^5$ (s/10^2)</th>
<th>$10^4$ (s/10^3)</th>
</tr>
</thead>
<tbody>
<tr>
<td>system</td>
<td>0.813</td>
<td>0.716</td>
<td>0.618</td>
<td>0.513</td>
<td>0.408</td>
</tr>
<tr>
<td>MSD radix</td>
<td>0.325</td>
<td>0.454</td>
<td>0.199</td>
<td>0.331</td>
<td>0.197</td>
</tr>
<tr>
<td>LSD radix</td>
<td>1.266</td>
<td>1.266</td>
<td>0.375</td>
<td>0.377</td>
<td>0.210</td>
</tr>
</tbody>
</table>