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

Consider only 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 (number of levels, not counting leaves)
Example: Merge Sort on $n = 3$
**Theorem.** Any decision tree that correctly sorts $n$ items must have height $\Omega(n \log n)$

**Proof.** All $n!$ permutations must appear as leaves. Therefore, if $h = \text{height of tree}$, then 

$$2^h \geq n! \implies h \geq \log_2 n!.$$ 

**Claim.** $\log_2 n! = n \log_2 n + O(n)$

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

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

therefore

\[ \int_{1}^{n} \log_2 x dx = n \log_2 n + O(n) \]

moreover

\[ \int_{1}^{n} \log_2 x dx \leq \sum_{i=1}^{n} \log_2 i \leq \int_{1}^{n} \log_2 x dx + \log_2 n \]

QED
Bucket Sort (aka Counting Sort)

Let $\Delta = \{0, \ldots, m - 1\}$

input: $a_1, \ldots, a_n \in \Delta$

initialize $T[j] \leftarrow \text{“empty list”} \ (j = 0 \ldots m - 1)$

for $i \leftarrow 1$ to $n$ do

\[ T[a_i] \leftarrow T[a_i] \parallel a_i \]

output $T[0] \parallel T[1] \parallel \cdots \parallel T[m - 1]$

Running time: $O(m + n)$

Note:

• this is a “stable” sort
Lexicographic Sort (1)

input: $A_1, \ldots, A_n \in \Delta^t$

for $j \leftarrow t$ down to 1 do
    bucket sort the $A_i$’s using $j$th entry as the “sort key”

Correctness: follows from stability of Bucket Sort

Running time: $O(nt + mt)$

Improvements:

- reduce running time to $O(nt + m)$
- handle variable length inputs
Lexicographic Sort (2) – variable length inputs

Input: $A_1, \ldots, A_n \in \Delta^*$, where $t_i := |A_i| > 0$, $t_{\text{max}} := \max \{t_i\}$, $N := \sum_i t_i$

Step 1: for $j = 1 \ldots t_{\text{max}}$, create a list $L[j]$ of all $A_i$’s of length $j$

Step 2: bucket sort so that ties get broken in favor of shorter strings:

$L \leftarrow \text{empty list}$

for $j \leftarrow t_{\text{max}}$ down to 1 do

$L \leftarrow L[j] \parallel L$

bucket sort $L$ using $j$th component as the “sort key”

Running time: $O(N + t_{\text{max}}m)$
Lexicographic Sort (3) – faster

Want to replace the $O(t_{\text{max}}m)$ term by $O(m)$.

- for large alphabets, this term could dominate

We spend most of our time looking at empty buckets

Visualize each $A_i$ as a row of characters in a 2D table

For each $j$, we want a sorted list of the characters that appear in the $j$th column of the table

Idea: use bucket sort (again!)
Input: $A_1, \ldots, A_n \in \Delta^*$, where $t_i := |A_i| > 0$, $t_{\text{max}} := \max \{t_i\}$, $N := \sum_i t_i$

Step 1: for $j = 1 \ldots t_{\text{max}}$, create a list $L[j]$ of all $A_i$'s of length $j$

Step 2: create a list of $N$ pairs $(j, a_{ij})$, where $a_{ij}$ is the $j$th component of $A_i$ [Time = $O(N)$]

Step 3: sort pairs lexicographically — Bucket Sort twice, first in the second component ($m$ buckets), and then in the first component ($t_{\text{max}}$ buckets) [Time = $O(N + m)$]

The output of this step looks like:

$(1, a), (1, c), (1, c), (1, d), (2, a), (2, c), (3, b), (3, b), (3, c), \ldots$

So we can read of the information we want
Step 4: run lex sort as before, except that we use the data from step 3 to ignore empty buckets

\[ L \leftarrow \text{empty list} \]
for \( j \leftarrow t_{\text{max}} \) down to 1 do
\[ L \leftarrow L[j] \parallel L \]
bucket sort \( L \) using \( j \)th component as the “sort key”, ignoring empty buckets

Running Time Analysis

The running time of loop iteration \( j \) is proportional to the number of pairs \((j, a_{ij})\)

The total cost is proportional to the total number of pairs, which is \( N \)
Putting it all together: total running time is $O(N + m)$

For constant $m$, or $m = O(N)$, this is linear in the input size

Does not contradict the sorting lower bound