Honors Algorithms
G22.3520-001 Fall 2006

Lecture 7
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
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 $h =$ height of tree, then

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

**Claim.** $\log_2 n! \sim n \log_2 n$

**Detour: 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
\[ \int_{1}^{n} \log_2 x \, dx \leq \sum_{i=1}^{n} \log_2 i \leq \int_{1}^{n} \log_2 x \, dx + \log_2 n \]
Moreover,
\[ \int_{1}^{n} \log_2 x \, dx = n \log_2 n + O(n) \]
QED
Bucket 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: $O(m + n)$

Notes:

- constant-time init trick does not work here
- 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)

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 $= O(N + m)$]
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
Divide and Conquer: a (somewhat) general theorem

The setup: a recursive algorithm that on inputs of size $n \geq n_0$, recursively solves

- $\leq a$ smaller sub-problems,
- each of size $\leq n/b + c$,
- with a “local” running time $\leq dn^e$

where $n_0, a, b, c, d, e$ are constants
Recursion tree analysis

At level 1, size $\leq n/b + c$

At level 2, size $\leq n/b^2 + c/b + c$

\ldots

At level $j$,

size $\leq n/b^j + c/b^{j-1} + \cdots + c/b + c$

$\leq n/b^j + C_1$,

where $C_1 := c/(1 - 1/b)$

At level $j$, there are $\leq a^j$ nodes
Set \( k := \lceil \log_b n \rceil \), so \( n \leq b^k < bn \)

At level \( k \), all sizes are \( \leq 1 + C_1 \), and we can ignore all nodes at levels \( k + 1, k + 2, \ldots \) (their contribution to the total cost is at most a constant times the sum of costs at level \( k \))

Let \( w = \text{sum of costs at levels } 0, \ldots, k \)

For each \( j = 0 \ldots k \), sum of costs at level \( j \) is

\[
\leq a^j \cdot d(n/b^j + C_1)^e \\
\leq C_2 a^j (n/b^j)^e \\
= C_2 n^e (a/b^e)^j
\]
Therefore,

\[ w \leq C_2 n^e \sum_{j=0}^{k} \alpha^j, \]

where \( \alpha := \alpha/b^e \)

**Case 1: \( \alpha < 1 \)**

\[ \sum_{j=0}^{\infty} \alpha^j = 1/(1 - \alpha) \implies w \leq (C_2/(1 - \alpha)) n^e \]

Total running time = \( O(n^e) \)

**Case 2: \( \alpha = 1 \)**

\[ \sum_{j=0}^{k} \alpha^j = (k + 1) \implies w \leq C_2 (k + 1) n^e \]

Total running time = \( O(n^e \log n) \)
Case 3: $\alpha > 1$

$$\sum_{j=0}^{k} \alpha^j = \frac{\alpha^{k+1} - 1}{\alpha - 1}$$

and so

$$w \leq C_3 n^e \alpha^k = C_3 n^e \alpha^k / (b^k)^e \leq C_3 \alpha^k$$

$$\leq C_3 \alpha^{\log_b n + 1} = C_3 \alpha \cdot \alpha^{\log_b n}$$

$$= C_3 \alpha \cdot b^{\log_b \alpha \cdot \log_b n}$$

$$= C_3 \alpha \cdot n^{\log_b \alpha}$$

Total running time $= O(n^{\log_b \alpha})$
Summarizing — the “Master Theorem”

Let $f := \log_b a$

**Case 1:** $e > f \implies O(n^e)$

**Case 2:** $e = f \implies O(n^e \log n)$

**Case 3:** $e < f \implies O(n^f)$
Application: faster multiplication

Problem: multiply two $n$-bit integers

An “$n$-bit integer” is an integer $a$ such that $0 \leq a < 2^n$

An $n$-bit integer can be represented using an array of $n$ bits (although in practice, one packs several bits into a “word”)

The sum of two $n$-bit integers is an $(n + 1)$-bit integer, and can be computed in time $O(n)$

The product of two $n$-bit integers is a $(2n)$-bit integer, and can be computed in time $O(n^2)$
Karatsuba’s multiplication algorithm

Input: two $n$-bit integers, $a$ and $b$

If $n$ is “very small”, use the naive algorithm

Otherwise, divide each number into two pieces:

\[
a = a_1 2^k + a_0
\]

\[
b = b_1 2^k + b_0,
\]

where $k := \lfloor n/2 \rfloor$

<table>
<thead>
<tr>
<th>$a$:</th>
<th>$a_1$</th>
<th>$a_0$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$b$:</td>
<td>$b_1$</td>
<td>$b_0$</td>
</tr>
</tbody>
</table>
\[ ab = a_1 b_1 2^{2k} + (a_1 b_0 + a_0 b_1) 2^k + a_0 b_0 \]
If we recursively compute the four sub-products $a_1 b_1, a_1 b_0, a_0 b_1, a_0 b_0$, we get another $O(n^2)$ algorithm

- $e = 1, f = \log_2 4 = 2$, Case 3 of Master Theorem

Better idea:

- Compute $A \leftarrow a_1 + a_0, B \leftarrow b_1 + b_0$
- Recursively compute three products:
  \( H \leftarrow a_1 b_1, \ L \leftarrow a_0 b_0, \ F \leftarrow AB \)
- Observe: $F = a_1 b_1 + a_1 b_0 + a_0 b_1 + a_0 b_0$
- Thus, we can compute $M \leftarrow F - (H + L)$, which is $a_1 b_0 + a_0 b_1$, and $P \leftarrow H2^{2k} + M2^k + L$, which is $ab$
Now apply Master Theorem: $e = 1,$
\[ f = \log_2 3 \approx 1.585 \]

Case 3: running time is $O(n^{\log_2 3})$

Notes:

- Not the fastest method: using the Fast Fourier Transform, one can multiply two $n$-bit integers in time $O(n \log n \log \log n)$
- For $n$ (roughly) in the range 500–10,000, Karatsuba is the fastest
- You use it every time you buy something from amazon.com, or use ssh — it’s used to implement public-key cryptosystems