Handouts: mergesort.py
Readings: CLRS Chaps 1-4, 11.1, 11.2
Python cost model
docdist1...docdist6

Web: mergesort.py, lecture notes

Admin: HW #1 will be posted today
laptop loaner program
new students?

Outline:
- Docdist review
- Asymptotic notation
- Mergesort: Divide & Conquer
  - Code
  - Analysis/Recurrences
  - Timing Experiments

Document Distance Review (Bobsey vs. Lewis)

<table>
<thead>
<tr>
<th>v1</th>
<th>initial</th>
<th>secs</th>
</tr>
</thead>
<tbody>
<tr>
<td>v2</td>
<td>profiled</td>
<td>194</td>
</tr>
<tr>
<td>v3</td>
<td>concatenate/extend</td>
<td>84</td>
</tr>
<tr>
<td>v4</td>
<td>dictionary instead of lists</td>
<td>41</td>
</tr>
<tr>
<td>v5</td>
<td>translate &amp; split</td>
<td>13</td>
</tr>
<tr>
<td>v6</td>
<td>merge-sort</td>
<td>6</td>
</tr>
<tr>
<td>(v7?)</td>
<td>no sorting!</td>
<td>&lt;1</td>
</tr>
</tbody>
</table>

(\text{Even though sorting is not necessary, it is very worthwhile to look at, so we shall...})
Sorting Problem: Given a list of $n$ comparable objects, rearrange them into increasing (nondecreasing) order.

Input sizes:
Time gets larger as inputs do.
Parameterize size with one or more measures ($n, m, \ldots$)
There are many inputs of a given size.

$T(n) = \text{worst-case running time on an input of size } n$

$$T(n) = \max_{(\text{inputs } x) \text{ of size } n} \left[ \text{running time on } x \right]$$

For insertion sort (ref docdist code, & CLRS §2.1)

$T(n) \approx \text{const} \cdot n^2$ (due to doubly-nested loops)

How to be precise about such things?
when * we don’t care about $T(n)$ for small $n$
* " " " " " const. factors
  (different computers, interpreted/compiled, etc...)

While running time might be

$$4n^2 + 22n - 12 \text{ microseconds}$$

we only care about high-order term ($4n^2$)
but without constant ($n^2$)

since other terms are negligible (relatively) as $n$ gets large.
"big oh" notation

We say
\[ T(n) \text{ is } \Theta(g(n)) \]
if
\[ \exists n_0 \quad \exists c \quad 0 \leq T(n) \leq c \cdot g(n) \text{ for all } n \geq n_0 \]

Example: \(4n^2 + 22n - 12\) is \(\Theta(n^2)\) since \(0 \leq i \leq 26n^2\) for \(n \geq 1\).

write \(4n^2 + 22n - 12 = \Theta(n^2)\) (but not reverse \(\Theta\) always on right)

Big Omega:
\[ T(n) = \Omega(g(n)) \]
if \((\exists n_0) (\exists c) 0 \leq c \cdot g(n) \leq T(n) \text{ for all } n \geq n_0\)

\[ 4n^2 + 22n - 12 = \Omega(n^2) \quad [c=1, n_0=1] \]

Big Theta:
\[ T(n) = \Theta(g(n)) \text{ iff } T(n) = \Omega(g(n)) \& T(n) = \Theta(g(n)) \]

\(g(n)\) is high-order term in \(T(n)\) (up to constant)

\(\therefore T(n) = 4n^2 + 22n - 12 = \Theta(n^2)\)

For insertion sort, \(T(n) = \Theta(n^2)\)

(\(\approx\) if you double input size, running time goes up \(4x\))

Can we do better? Yes!
Divide/Conquer/Combine paradigm
aka "Divide & Conquer"
by example: mergesort

6.006
Rivest
L2.4
9/9/08

---

Divide

\[ A \]

divide

\[ L \quad R \]

input array of size \( n \)

2 arrays of size \( n/2 \)

conquer recursively

\[ L \quad R \]

Sort

\[ L \quad R \]

2 sorted arrays of size \( n/2 \)

combine

\[ \text{merge} \]

sorted \( A \)

Sorted array of size \( n \)

Show code (handout): merge_sort

merge ("two finger algorithm")

Ex. merge

\[
\begin{array}{ccccccc}
5 & 4 & 7 & 3 & 6 & 1 & 9 \quad 2 \\
3 & 4 & 5 & 7 & 1 & 2 & 6 \quad 9 \\
\end{array}
\]

(show merge)

\[
1 \quad 2 \quad 3 \quad 4 \quad 5 \quad 6 \quad 7 \quad 9
\]
Analysis:

Running time of merge on two inputs of size \( n/2 \) is \( cn \), for some \( c \).

Let \( T(n) \) = running time of mergesort on inputs of size \( n \).

\[
T(n) = c_1 + \frac{2}{n} T\left(\frac{n}{2}\right) + cn
\]

(only keep high-order terms)

\[
= cn + 2 \left( c \cdot \left(\frac{n}{2}\right) + 2 \left( c \cdot \left(\frac{n}{2}\right) + \ldots \right) \right)
\]

\[
T(n) = cn \cdot (\lg(n) + 1)
\]

\( = \Theta(n \lg n) \)

Ref: CLRS
Chapter 4
Experimental Results

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Complexity</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>Insertion-sort</td>
<td>Θ(n²)</td>
<td>6.006</td>
</tr>
<tr>
<td>Merge-sort</td>
<td>Θ(n lg(n))</td>
<td>L2.6</td>
</tr>
<tr>
<td>&quot;sorted&quot; (built-in)</td>
<td>Θ(n lg(n))</td>
<td>1/9/08</td>
</tr>
</tbody>
</table>

**insertion_sort**

- test_insertion (2**12) ≈ 1 second
- Insertion sort takes ≈ 66 * n² nanoseconds
- ... test(test_insertion)...

**merge_sort**

- test_merge (2**17) ≈ 1.5 seconds
- Merge sort takes ≈ 701 * n lg n nanoseconds
- ... test(test_merge)...

**Sorted (built-in)**

- test_sorted (2**20) ≈ 1 second
- Sorted takes ≈ 55 * n lg n nanoseconds
- ... test(test_sorted)...

- Not quite linear, as lg(n) grows slowly, but "almost".

- Small constant for "sorted", since it is written in C, (13x speedup ?) but asymptotics same as for mergesort.
When is mergesort (in Python) better than insertion-sort in C?

701 \, n \lg(n) \text{ nanoseconds}

5 \, n^2 \text{ nanoseconds} \quad (5 \approx 66/13)

crossover: \quad 5n^2 \geq 701 \, n \lg(n)

at \quad n \geq 1500

Mergesort wins for \quad n \geq 1500

Better algorithm much more valuable than hardware or compiler, even for modest \, n.

[Note: hybrid approach: use insertion sort if \, n \leq 1500

merge-sort if \, n \geq 1500 ]

- Python Cost Model - Similar experiments on other operations
  - uses timing.py to "fit" formula to data
  - (code might not be so readable...)
  - look at chart...

- **Homework:** \quad S = \text{set}([1,2,3]) \quad \text{set data type}
  
  T = \text{set}([1,2,4,9])
  
  \quad S \, \text{intersection}(T) = \text{set}([1,2])

  \[ S \cap T \]

  |S|, |T|, and |S \cap T|

  running time may depend on

  \quad \text{figure it out!}

//end of first module