New Room: 32-123
Handouts: mergesort.py
Readings: CLRS Chaps 1-4, 11.1, 11.2
Python cost model
docdist1...docdist6
Web: mergesort.py, lecture notes
Admin: HW 11 will be posted today

leapfrog 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>
<th>Θ(n^2)</th>
</tr>
</thead>
<tbody>
<tr>
<td>v2</td>
<td>profiled</td>
<td>194</td>
<td>Θ(n^2)</td>
</tr>
<tr>
<td>v3</td>
<td>concatenate extend</td>
<td>84</td>
<td>Θ(n^2)</td>
</tr>
<tr>
<td>v4</td>
<td>dictionaries instead of labs</td>
<td>41</td>
<td>Θ(n^2)</td>
</tr>
<tr>
<td>v5</td>
<td>translate &amp; split</td>
<td>13</td>
<td>Θ(n^2)</td>
</tr>
<tr>
<td>v6</td>
<td>merge-sort</td>
<td>6</td>
<td>Θ(n log n)</td>
</tr>
<tr>
<td>v7</td>
<td>no sorting!</td>
<td>&lt; 1</td>
<td>Θ(n)</td>
</tr>
</tbody>
</table>

(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.

\[
T(n) = \text{worst-case running time on an input of size } n = \max \left[ \text{running time on } x \right]
\]
\((\text{of size } n)\)

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

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

How to be precise about such things?

- when we don't care about \(T(n)\) for small \(n\)
- "" "" constant factors
  (different computers, interpreted/compiled, etc...)

While running time might be

\[4n^2 + 22n - 12\] 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.

AE: though in practice constants can vary
"big oh" notation

We say $T(n)$ is $O(g(n))$

if

$\exists n_0$

$\exists c$

st. $0 \leq T(n) \leq c \cdot g(n)$ for all $n \geq n_0$

Example: $4n^2 + 22n - 12$ is $O(n^2)$

since $0 \leq i \leq 26n^2$ for $n \geq 1$.

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

lower bound

Big Omega:

$T(n) = \Omega(g(n))$

if $(\exists n_0)(\exists c)\ 0 \leq c \cdot g(n) \leq T(n)$ for all $n \geq n_0$

$bith$

Big Theta:

$T(n) = \Theta(g(n))$ iff $T(n) = O(g(n)) \land T(n) = \Omega(g(n))$

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

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

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

($\propto$ if you double input size, running time goes up $4x.$)

Can we do better? Yes!

than insertion sort
Divide/Conquer/Combine paradigm
aka "Divide & Conquer"
by example: mergesort

input array of size n

divide

L   R

2 arrays of size n/2

Conquer recursively

Sort

Sort

L   R

2 sorted arrays of size n/2

Combine

merge

sorted A

Sorted array of size n

Show code (handout): merge_sort

merge ("two finger algorithm")

Ex. merge

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

(show merge)
Experimental Results

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Complexity</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Insertion-sort</td>
<td>$\Theta(n^2)$</td>
<td></td>
</tr>
<tr>
<td>Merge-sort</td>
<td>$\Theta(n \log(n))$</td>
<td>Rivest 1.2.6</td>
</tr>
<tr>
<td>&quot;sorted&quot; (built-in)</td>
<td>$\Theta(n \log(n))$</td>
<td>9/9/08</td>
</tr>
</tbody>
</table>

---

**insertion_sort**

test_insertion($2^{*12}$) $\approx$ 1 second

insertion_sort takes $\approx$ 66 $\cdot$ $n^2$ nanoseconds

... test(test_insertion)...

**merge_sort**

test_merge($2^{*17}$) $\approx$ 1.5 seconds

merge_sort takes $\approx$ 70 $\cdot$ $n \log(n)$ nanoseconds

... test(test_merge)...

**Sorted (built-in)**

test_sorted($2^{*20}$) $\approx$ 1 second

sorted takes $\approx$ 55 $\cdot$ $n \log(n)$ nanoseconds

... test(test_sorted)...

- Not quite linear, as $\log(n)$ grows slowly, but "almost".

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

AE: Too easy to say "C" is faster than python. Even "C" codes can vary quite a bit. I tried it in matlab and it was considerably...
Analysis:

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

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

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

\((T(1)=c) = 2T\left(\frac{n}{2}\right) + c \cdot n \quad \text{(only keep high-order terms)}\)

\[
= cn + 2(c \cdot \frac{n}{2}) + 2(c \cdot \frac{n}{4}) + \cdots
\]

\[
T(n) = c \cdot n \cdot (\log(n) + 1)
\]

\(= \Theta(n \log n)\)

Ref:

CLRS
Chapter 4
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 5 n^2 \gg 701 n \lg n

\text{ at } n \geq 1500

Mergesort wins for \( n \geq 1500 \)

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 \( n > 1500 \)]

• 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:
  \( S = \text{set}([1,2,3]) \) \hspace{1cm} \text{set data type}
  \( T = \text{set}([1,2,4,9]) \)
  \( S, \text{intersection}(T) = \text{set}([1,2]) \)

\begin{align*}
S \cap T &\quad \text{running time may depend on}
S \cup T &\quad |S|, |T|, \text{ and } |S \cap T|
\end{align*}

figure it out! \quad \text{(FOR READER MAY BE HARD?)}

// end of first module