Lecture 2

More logistics

Divide-and-conquer, MergeSort, and Big-O notation
Some questions from last time

• What’s the best way to take notes?
  • I will post slides ahead of time.
  • Sometimes the slides will refer to explanations on the board – I encourage you to take notes for that stuff.

• What’s with the waitlist?
  • We got another TA! (Danny Wright). Enrollment will be opening up (still limited, but everyone who was on the waitlist at 8am this morning will get a spot).
Logistics some more!

• As before: [http://web.stanford.edu/class/cs161/](http://web.stanford.edu/class/cs161/)
• There will be office hours! And recitation sections!

The deal with sections:

• Held on Mondays and Tuesdays: go over content from previous week, with example problems!
• Don’t need to sign up, go to any/either section!
• Problems/solutions from both sections will be posted!
Homework! We’re gonna have it!

- Assignment will be posted **FRIDAY.** (3pm)
- Due the next **FRIDAY.** (3pm)
- See the **COURSE WEBSITE** for HW guidelines.
- Solutions most the following **MONDAY.**

<table>
<thead>
<tr>
<th></th>
<th>Monday</th>
<th>Tuesday</th>
<th>Wednesday</th>
<th>Thursday</th>
<th>Friday</th>
</tr>
</thead>
<tbody>
<tr>
<td>HW</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td><strong>ASSIGNED</strong></td>
</tr>
<tr>
<td>HW</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td><strong>DUE</strong></td>
</tr>
<tr>
<td>HW</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td><strong>RELEASED</strong></td>
</tr>
<tr>
<td>HW</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td><strong>RETURNED</strong> (target)</td>
</tr>
</tbody>
</table>
Last time

Philosophy

- Algorithms are awesome and powerful!
- Algorithm designer’s question: Can I do better?
- Interplay between rigor and intuition.

Technical content

- Karatsuba integer multiplication
- Example of “Divide and Conquer”
- Not-so-rigorous analysis
Today

• Sorting
• Return of divide-and-conquer with Merge Sort
• Skills:
  • Analyzing correctness of iterative and recursive algorithms.
  • Analyzing running time of recursive algorithms.

• How do we measure the runtime of an algorithm?
  • Worst-case analysis
  • Asymptotic Analysis
Sorting

- Important primitive
- For today, we’ll pretend all elements are distinct.
Benchmark: insertion sort

• Say we want to sort: 6 4 3 8 5

• Insert items one at a time.

• How would we actually implement this?
Insertion sort example...

Start with the second element (the first element is sorted within itself...)

Pull “4” back until it’s in the right place.

Now look at “3”

Pull “3” back until it’s in the right place.

“8” is good...look at 5

(then fix 5 and we’re done)
Insertion sort pseudocode

Go one-at-a-time until things are in the right place.

Algorithm 1: `INSERTIONSORT(A)`

```plaintext
for i = 2 → length(A) do
    key ← A[i];
    j ← i - 1;
    while j > 0 and A[j] > key do
        j ← j - 1;
    A[j + 1] ← key;
```

- (Discussion on board)
Insertion sort: running time

Algorithm 1: INSERTIONSORT(A)

\[
\text{for } i = 2 \rightarrow \text{length}(A) \text{ do} \\
\quad \text{key } \leftarrow A[i]; \\
\quad j \leftarrow i - 1; \\
\quad \text{while } j > 0 \text{ and } A[j] > \text{key} \text{ do} \\
\quad \quad A[j + 1] \leftarrow A[j]; \\
\quad \quad j \leftarrow j - 1; \\
\quad \quad A[j + 1] \leftarrow \text{key}; \\
\]

Running time is about \(n^2\)
Insertion sort: correctness

• **Maintain a loop invariant.**

• **Initialization**: the loop invariant holds before the first iteration.

• **Maintenance**: If it is true before the t’th iteration, it will be true before the (t+1)’st iteration

• **Termination**: It is useful to know that the loop invariant is true at the end of the last iteration.

(This is proof-of-correctness by induction)
Insertion sort: correctness

- **Loop invariant**: At the start of the t’th iteration (of the outer loop), the first t elements of the array are sorted.
- **Initialization**: At the start of the first iteration, the first element of the array is sorted. ✓
- **Maintenance**: By construction, the point of the t’th iteration is to put the (t+1)’st thing in the right place.
- **Termination**: At the start of the (len(A) + 1)’st iteration (aka, at the end of the algorithm), the first len(A) items are sorted. ✓
To summarize

**InsertionSort** is an algorithm that correctly sorts an arbitrary n-element array in time about $n^2$.

Can we do better?
Can we do better?

- **MergeSort**: a *divide-and-conquer* approach
- Recall from last time:

**Diagram:**

- **Big problem**
  - **Smaller problem**
    - **Yet smaller problem**
    - **Yet smaller problem**
  - **Smaller problem**
    - **Yet smaller problem**
    - **Yet smaller problem**
  - **Smaller problem**
    - **Yet smaller problem**
    - **Yet smaller problem**
  - **Smaller problem**
    - **Yet smaller problem**
    - **Yet smaller problem**
  - **Smaller problem**
    - **Yet smaller problem**
    - **Yet smaller problem**

Divide and Conquer:
MergeSort

\[
\begin{array}{cccccccc}
6 & 4 & 3 & 8 & 1 & 5 & 2 & 7 \\
\end{array}
\]

Recursive magic!

\[
\begin{array}{cccc}
3 & 4 & 6 & 8 \\
\end{array}
\]

Recursive magic!

\[
\begin{array}{cccc}
1 & 5 & 2 & 7 \\
\end{array}
\]

MERGE!

\[
\begin{array}{cccccccc}
1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 \\
\end{array}
\]

(MERGE pseudocode + analysis on board)

(A good exercise: how would I do this in place?)
MergeSort Pseudocode

MERGESORT(A):

• n = length(A)

• if n ≤ 1:  
  • return A  
  If A has length 1,  
  It is already sorted!

• L = MERGESORT(A[1 : n/2])  
  Sort the left half

• R = MERGESORT(A[n/2+1 : n ])
  Sort the right half

• return MERGE(L,R)  
  Merge the two halves
What actually happens when we run MergeSort?

• (on board)
Schematic of recursive calls

Original sequence

Sorted sequence!
Two questions

• Does this work?
• Is it fast?
It works  

Let’s assume $n = 2^t$

**Invariant:**

“In every recursive call, **MERGESORT** returns a sorted array.”

- **Base case** $(n=1)$: a 1-element array is always sorted.
- **Maintenance**: Suppose that $L$ and $R$ are sorted. Then **MERGE($L$,$R$)** is sorted.
- **Termination**: “In the top recursive call, **MERGESORT** returns a sorted array.”

The maintenance step needs more details!! Why is this statement true?

Not technically a “loop invariant,” but a “recursion invariant,” that should hold at the beginning of every recursive call.

- $n = \text{length}(A)$
- **if** $n \leq 1$:
  - **return** $A$
- $L = \text{MERGESORT}(A[1 : n/2])$
- $R = \text{MERGESORT}(A[n/2+1 : n])$
- **return** **MERGE**(L,R)
It’s fast Let’s keep assuming $n = 2^t$

**CLAIM:**

MERGESORT requires at most $6n \log(n) + 6n$ operations to sort $n$ numbers.

Before we see why...
How does that compare to the $\approx n^2$ operations of INSERTIONSORT?
n log(n) vs n^2

• log(n) : how many times do you need to divide n by 2 in order to get down to 1?

<table>
<thead>
<tr>
<th>n</th>
<th>log(n)</th>
</tr>
</thead>
<tbody>
<tr>
<td>32</td>
<td>5</td>
</tr>
<tr>
<td>16</td>
<td>4</td>
</tr>
<tr>
<td>8</td>
<td>3</td>
</tr>
<tr>
<td>4</td>
<td>2</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>64</td>
<td>6</td>
</tr>
<tr>
<td>32</td>
<td>5</td>
</tr>
<tr>
<td>16</td>
<td>4</td>
</tr>
<tr>
<td>8</td>
<td>3</td>
</tr>
<tr>
<td>4</td>
<td>2</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>64</td>
<td>6</td>
</tr>
</tbody>
</table>

log(32) = 5 log(64) = 6

log(128) = 7 log(256) = 8 log(512) = 9

Moral: log(n) grows very slowly with n.

log(number of particles in the universe) < 280
### n log(n) vs $n^2$

**Continued**

<table>
<thead>
<tr>
<th>n</th>
<th>n log(n)</th>
<th>$n^2$</th>
</tr>
</thead>
<tbody>
<tr>
<td>8</td>
<td>24</td>
<td>64</td>
</tr>
<tr>
<td>16</td>
<td>64</td>
<td>256</td>
</tr>
<tr>
<td>32</td>
<td>160</td>
<td>1024</td>
</tr>
<tr>
<td>64</td>
<td>384</td>
<td>4096</td>
</tr>
<tr>
<td>128</td>
<td>896</td>
<td>16384</td>
</tr>
<tr>
<td>256</td>
<td>2048</td>
<td>65536</td>
</tr>
<tr>
<td>512</td>
<td>4608</td>
<td>262144</td>
</tr>
<tr>
<td>1024</td>
<td>10240</td>
<td>1048576</td>
</tr>
</tbody>
</table>
It’s fast!

CLAIM:

MERGESORT requires at most \(6n \log(n) + 6n\) operations to sort \(n\) numbers.

As \(n\) grows, that’s much faster than the \(\approx n^2\) operations of INSERTIONSORT!
Analysis

T(n) = time to run MERGESORT on a list of size n

This is called a recurrence relation: it describes the running time of a problem of size n in terms of the running time of smaller problems.

\[ T(n) = T(n/2) + T(n/2) + T(\text{MERGE}) = 2T(n/2) + 6n \]

T(\text{MERGE} two lists of size n/2) is the time to do:

\begin{itemize}
  \item 3 variable assignments (counters ← 1)
  \item n comparisons
  \item n more assignments
  \item 2n counter increments
\end{itemize}

So that’s

\[ 2T(\text{assign}) + n T(\text{compare}) + n T(\text{assign}) + 2n T(\text{increment}) \]

or \(4n + 2\) operations

Let’s say \(T(\text{MERGE} \text{ of size } n/2) \leq 6n\) operations

We will see later how to analyze recurrence relations like these automagically...but today we’ll do it from first principles.
## Recursion tree

<table>
<thead>
<tr>
<th>Level</th>
<th># problems</th>
<th>Size of each problem</th>
<th>Amount of work at this level (just MERGEing)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>n</td>
<td>6n</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td>n/2</td>
<td>6n</td>
</tr>
<tr>
<td>2</td>
<td>4</td>
<td>n/4</td>
<td>6n</td>
</tr>
<tr>
<td>t</td>
<td>$2^t$</td>
<td>$n/2^t$</td>
<td>6n</td>
</tr>
<tr>
<td>log(n)</td>
<td>n</td>
<td>1</td>
<td>6n</td>
</tr>
</tbody>
</table>
Total runtime...

• 6n steps per level, at every level

• log(n) + 1 levels

• $6n \log(n) + 6n$ steps total

That was the claim!
A few reasons to be grumpy

• Sorting


should take zero steps...

• What’s with this $T(MERGE) < 6n$?
  • $2 + 4n < 6n$ is a loose bound.
  • Different operations don’t take the same amount of time.
Today

- Sorting
- Return of divide-and-conquer with Merge Sort

**Skills:**
- Analyzing correctness of iterative and recursive algorithms.
- Analyzing running time of recursive algorithms.

- How do we measure the runtime of an algorithm?
  - Worst-case analysis
  - Asymptotic Analysis
NOTE!!! WE ACTUALLY STOPPED HERE IN CLASS ON WEDNESDAY!
WE’LL PICK UP AFTER THIS NEXT TIME...
Worst-case analysis

• In this class, we will focus on **worst-case analysis**

  ![Algorithm designer]

  **Here is my algorithm!**

  Algorithm:
  - Do the thing
  - Do the stuff
  - Return the answer

  ![Here is an input!]

  • **Pros:** very strong guarantee
  • **Cons:** very strong guarantee

Sorting a sorted list should be fast!!
Big-O notation

• What do we mean when we measure runtime?
  • We probably care about wall time: how long does it take to solve the problem, in seconds or minutes or hours?

• This is heavily dependent on the programming language, architecture, etc.

• These things are very important, but are not the point of this class.

• We want a way to talk about the running time of an algorithm, independent of these considerations.
Remember this slide?

<table>
<thead>
<tr>
<th>n</th>
<th>n log(n)</th>
<th>n^2</th>
</tr>
</thead>
<tbody>
<tr>
<td>8</td>
<td>24</td>
<td>64</td>
</tr>
<tr>
<td>16</td>
<td>64</td>
<td>256</td>
</tr>
<tr>
<td>32</td>
<td>160</td>
<td>1024</td>
</tr>
<tr>
<td>64</td>
<td>384</td>
<td>4096</td>
</tr>
<tr>
<td>128</td>
<td>896</td>
<td>16384</td>
</tr>
<tr>
<td>256</td>
<td>2048</td>
<td>65536</td>
</tr>
<tr>
<td>512</td>
<td>4608</td>
<td>262144</td>
</tr>
<tr>
<td>1024</td>
<td>10240</td>
<td>1048576</td>
</tr>
</tbody>
</table>
Change $n \log(n)$ to $5n \log(n)$.

<table>
<thead>
<tr>
<th>n</th>
<th>$5n \log(n)$</th>
<th>$n^2$</th>
</tr>
</thead>
<tbody>
<tr>
<td>8</td>
<td>120</td>
<td>64</td>
</tr>
<tr>
<td>16</td>
<td>320</td>
<td>256</td>
</tr>
<tr>
<td>32</td>
<td>800</td>
<td>1024</td>
</tr>
<tr>
<td>64</td>
<td>1920</td>
<td>4096</td>
</tr>
<tr>
<td>128</td>
<td>4480</td>
<td>16384</td>
</tr>
<tr>
<td>256</td>
<td>10240</td>
<td>65536</td>
</tr>
<tr>
<td>512</td>
<td>23040</td>
<td>262144</td>
</tr>
<tr>
<td>1024</td>
<td>51200</td>
<td>1048576</td>
</tr>
</tbody>
</table>

As $n$ gets large, I’d even take runtime 100 $n \log(n)$ over $n^2$. 
Asymptotic Analysis
How does the running time scale as $n$ gets large?

One algorithm is “faster” than another if its runtime grows more “slowly” as $n$ gets large.

**Pros:**
- Abstracts away from hardware- and language-specific issues.
- Makes algorithm analysis much more tractable.

**Cons:**
- Only makes sense if $n$ is large (compared to the constant factors).
O(...) means an upper bound

- We say “T(n) is O(f(n))” if f(n) grows at least as fast as T(n) as n gets large.

- Formally,

\[
T(n) = O(f(n)) \iff \exists c, n_0 > 0 \text{ s.t. } \forall n \geq n_0,\quad 0 \leq T(n) \leq c \cdot f(n)
\]

(Explanation of what all these symbols mean on board)
Parsing that...

\[ T(n) = O(f(n)) \]

\[ \iff \]

\[ \exists c, n_0 > 0 \text{ s.t. } \forall n \geq n_0, \]

\[ 0 \leq T(n) \leq c \cdot f(n) \]
Example 1

- $T(n) = n$, $f(n) = n^2$.
- $T(n) = O(f(n))$

Why do we need $c$ in the definition?

$T(n) = O(f(n)) \iff \exists c, n_0 > 0 \text{ s.t. } \forall n \geq n_0, 0 \leq T(n) \leq c \cdot f(n)$

(formal proof on board)
Example 2  

(Need c but not really $n_0$)

- $g(n) = 2$, $f(n) = 1$.
- $g(n) = O(f(n))$ (and also $f(n) = O(g(n))$)

\[ T(n) = O(f(n)) \iff \exists c, n_0 > 0 \text{ s.t. } \forall n \geq n_0, \]
\[ 0 \leq T(n) \leq c \cdot f(n) \]

Diagram:
- $g(n)$
- $f(n)$
- $2.1 \cdot f(n)$
- $n_0 = 1$
Example 3 (Need both $c$ and $n_0$)

- $f(n) = 1$, $g(n)$ as below.
- $g(n) = O(f(n))$ (and also $f(n) = O(g(n))$)

$T(n) = O(f(n)) \iff \exists c, n_0 > 0 \text{ s.t. } \forall n \geq n_0, 0 \leq T(n) \leq c \cdot f(n)$
Examples 4 and 5

• All degree k polynomials are $O(n^k)$
• For any $k \geq 1$, $n^k$ is not $O(n^{k-1})$
Take-away from examples

• To prove $T(n) = O(f(n))$, you have to come up with $c$ and $n_0$ so that the definition is satisfied.

• To prove $T(n)$ is **NOT** $O(f(n))$, one way is by contradiction:
  • Suppose that someone gives you a $c$ and an $n_0$ so that the definition is satisfied.
  • Show that this someone must by lying to you by deriving a contradiction.
Ω(...) means a lower bound

• We say “T(n) is Ω(f(n))” if f(n) grows at most as fast as T(n) as n gets large.

• Formally,

\[ T(n) = O(f(n)) \quad \iff \quad \exists c, n_0 > 0 \text{ s.t. } \forall n \geq n_0, \]

\[ 0 \leq c \cdot f(n) \leq T(n) \]

Switched these!!
Parsing that...

\[ T(n) = O(f(n)) \]

\[ \iff \exists c, n_0 > 0 \text{ s.t. } \forall n \geq n_0, \]
\[ 0 \leq c \cdot f(n) \leq T(n) \]
\(\Theta(...)\) means both!

- We say “\(T(n)\) is \(\Theta(f(n))\)” if:

\[
T(n) = O(f(n)) \quad \text{AND} \quad T(n) = \Omega(f(n))
\]
Yet more examples

• $n^3 + 3n = O(n^3 - n^2)$
• $n^3 + 3n = \Omega(n^3 - n^2)$
• $n^3 + 3n = \Theta(n^3 - n^2)$

• $3^n$ is NOT $O(2^n)$
• $n \log(n) = \Omega(n)$
• $\log(n) = \Theta(2^{\log\log(n)})$
Some brainteasers

• Are there functions f, g so that \textbf{NEITHER} \( f = O(g) \) nor \( f = \Omega(g) \)?

• Are there \textit{non-decreasing} functions f, g so that the above is true?

• Define the n’th fibonacci number by \( F(0) = 1 \), \( F(1) = 1 \), \( F(n) = F(n-1) + F(n-2) \) for \( n > 2 \).
  • 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, ...

True or false:
  • \( F(n) = O(2^n) \)
  • \( F(n) = \Omega(2^n) \)
We’ll be using lots of asymptotic notation from here on out.

But we should always be careful not to abuse it.

In the course, (almost) every algorithm we see will be actually practical, without needing to take \( n \geq n_0 = 2^{10000000}. \)
Recap

• **Divide-and-conquer** paradigm
  • **Sorting:** Merge Sort

• **How do we measure the runtime of an algorithm?**
  • Worst-case analysis
  • Asymptotic Analysis

• **Next time:**
  • Integer multiplication part II
  • A more systematic approach to solving recurrences