Lecture 2

Divide-and-conquer, MergeSort, and Big-O notation
Announcements

• Homework!
  • HW1 will be released Friday.
  • It is due the following Friday.

• See the website for guidelines on homework, including:
  • Collaboration policy
  • Best practices/style guide
    • Will be posted by Friday!
Last time

Philosophy

• Algorithms are awesome and powerful!
• Algorithm designer’s question: Can I do better?

Technical content

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

Cast

Plucky the pedantic penguin
Lucky the lackadaisical lemur
Ollie the over-achieving ostrich
Siggi the studious stork
Today

• Things we want to know about algorithms:
  • Does it work?
  • Is it efficient?

• We’ll start to see how to answer these by looking at some examples of sorting algorithms.
  • InsertionSort
  • MergeSort

SortingHatSort not discussed
The plan

Part I: Sorting Algorithms
- InsertionSort: does it work and is it fast?
- MergeSort: does it work and is it fast?
- Skills:
  - Analyzing correctness of iterative and recursive algorithms.
  - Analyzing running time of recursive algorithms (part 1...more next time!)

Part II: 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.
I hope everyone did the pre-lecture exercise!

What was the mystery sort algorithm?

1. MergeSort
2. QuickSort
3. InsertionSort
4. BogoSort

```python
def mysteryAlgorithmOne(A):
    B = [None for i in range(len(A))]
    for i in range(len(B)):
        if B[i] == None or B[i] > x:
            j = len(B)-1
            while j > i:
                B[j] = B[j-1]
                j -= 1
            B[i] = x
            break
    return B

def MysteryAlgorithmTwo(A):
    for i in range(1,len(A)):
        current = A[i]
        j = i-1
        while j >= 0 and A[j] > current:
            j -= 1
        A[j+1] = current
```
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?
def InsertionSort(A):
    for i in range(1,len(A)):
        current = A[i]
        j = i-1
        while j >= 0 and A[j] > current:
            j -= 1
        A[j+1] = current
InsertionSort example

Start by moving A[1] toward the beginning of the list until you find something smaller (or can’t go any further):

6 4 3 8 5
4 6 3 8 5

Then move A[2]:

4 6 3 8 5
3 4 6 8 5

Then move A[3]:

3 4 6 8 5
3 4 6 8 5

Then move A[4]:

3 4 6 8 5
3 4 5 6 8

Then we are done!
Insertion Sort

1. Does it work?
2. Is it fast?
Empirical answers...

- Does it work?
  - You saw it worked on the pre-Lecture exercise.
- Is it fast?
  - IPython notebook lecture2_sorting.ipynb says:
Insertion Sort

1. Does it work?
2. Is it fast?

- The “same” algorithm can be faster or slower depending on the implementation...
- We are interested in how fast the running time scales with $n$, the size of the input.
Insertion Sort: running time

```python
def InsertionSort(A):
    for i in range(1, len(A)):
        current = A[i]
        j = i - 1
        while j >= 0 and A[j] > current:
            j -= 1
        A[j+1] = current
```

In the worst case, about \( n \) iterations of this inner loop

Running time scales like \( n^2 \)
Insertion Sort

1. Does it work?
2. Is it fast?

• Okay, so it’s pretty obvious that it works.

• HOWEVER! In the future it won’t be so obvious, so let’s take some time now to see how we would prove this rigorously.
Why does this work?

• Say you have a sorted list, \(3 4 6 8\), and another element \(5\).

• Insert \(5\) right after the largest thing that’s still smaller than \(5\). (Aka, right after \(4\)).

• Then you get a sorted list: \(3 4 5 6 8\).
So just use this logic at every step.

The first element, [6], makes up a sorted list.

So correctly inserting 4 into the list [6] means that [4,6] becomes a sorted list.

The first two elements, [4,6], make up a sorted list.

So correctly inserting 3 into the list [4,6] means that [3,4,6] becomes a sorted list.

The first three elements, [3,4,6], make up a sorted list.

So correctly inserting 8 into the list [3,4,6] means that [3,4,6,8] becomes a sorted list.

The first four elements, [3,4,6,8], make up a sorted list.

So correctly inserting 5 into the list [3,4,6,8] means that [3,4,5,6,8] becomes a sorted list.

YAY WE ARE DONE!
Recall: proof by induction

• Maintain a loop invariant.
• Proceed by induction.

• Four steps in the proof by induction:
  • Inductive Hypothesis: The loop invariant holds after the $i^{th}$ iteration.
  • Base case: the loop invariant holds before the 1$^{st}$ iteration.
  • Inductive step: If the loop invariant holds after the $i^{th}$ iteration, then it holds after the $(i+1)^{st}$ iteration.
  • Conclusion: If the loop invariant holds after the last iteration, then we win.

A loop invariant is something that should be true at every iteration.
Formally: induction

- **Loop invariant(i):** $A[:i+1]$ is sorted.

- **Inductive Hypothesis:**
  - The loop invariant(i) holds at the end of the $i^{th}$ iteration (of the outer loop).

- **Base case (i=0):**
  - Before the algorithm starts, $A[:1]$ is sorted. ✓

- **Inductive step:**

- **Conclusion:**
  - At the end of the $n-1^{st}$ iteration (aka, at the end of the algorithm), $A[:n] = A$ is sorted.
  - That’s what we wanted! ✓

The first two elements, [4,6], make up a sorted list.

The first two elements, [4,6], make up a sorted list.

So correctly inserting 3 into the list [4,6] means that [3,4,6] becomes a sorted list.
Aside: proofs by induction

• We’re gonna see/do/skip over a lot of them.
• I’m assuming you’re comfortable with them from CS103.
  • When you assume...
• If that went by too fast and was confusing:
  • Slides [there’s a hidden one with more info]
  • Lecture notes
  • Book
  • Office Hours

Make sure you really understand the argument on the previous slide!
To summarize

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

Can we do better?
The plan

• Part I: Sorting Algorithms
  • InsertionSort: does it work and is it fast?
  • MergeSort: does it work and is it fast?

• Skills:
  • Analyzing correctness of iterative and recursive algorithms.
  • Analyzing running time of recursive algorithms (part A)

• Part II: How do we measure the runtime of an algorithm?
  • Worst-case analysis
  • Asymptotic Analysis
Can we do better?

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

Code for the MERGE step is given in the Lecture2 notebook or the Lecture Notes.
MergeSort Pseudocode

**MERGESORT**(A):

- \( n = \text{length}(A) \)
- \( \textbf{if} \ n \leq 1: \) If A has length 1, It is already sorted!
  - \( \textbf{return} \ A \)
- \( L = \text{MERGESORT}(A[0 : n/2]) \) Sort the left half
- \( R = \text{MERGESORT}(A[n/2 : n]) \) Sort the right half
- \( \textbf{return} \ \text{MERGE}(L,R) \) Merge the two halves
What actually happens?

First, recursively break up the array all the way down to the base cases.

This array of length 1 is sorted!
Then, merge them all back up!

A bunch of sorted lists of length 1 (in the order of the original sequence).
Two questions

1. Does this work?
2. Is it fast?

Empirically:

1. Seems to.
2. Maybe?

IPython notebook says...
It works  Let’s assume \( n = 2^t \)

• Inductive hypothesis:
  
  “In every recursive call, 
  MERGESORT returns a sorted array.”

• Base case (\( n=1 \)): a 1-element array is always sorted.
• Inductive step: Suppose that \( L \) and \( R \) are sorted. Then \( \text{MERGE}(L,R) \) is sorted.
• Conclusion: “In the top recursive call, \( \text{MERGESORT} \) returns a sorted array.”

Fill in the inductive step! (Either do it yourself or read it in CLRS!)

Again we’ll use induction. This time with an invariant that will remain true after every recursive call.

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

CLAIM:

MERGESORT requires at most $11n (\log(n) + 1)$ operations to sort $n$ numbers.

What exactly is an “operation” here? We’re leaving that vague on purpose. Also I made up the number 11.

How does this compare to InsertionSort?

Scaling like $n^2$ vs scaling like $n\log(n)$?
Empirically

InsertionSort vs MergeSort

[Blue line] Less naive version of insertion sort
[Orange line] Not very slick implementation of mergeSort

This grows like $n^2$

This supposedly grows like $n \log(n)$
The constant doesn’t matter: eventually, $n^2 > 1111111 \cdot n \log(n)$
Quick log refresher

- **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>32</th>
<th>64</th>
<th>log(128) = 7</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>16</td>
<td>32</td>
<td>log(256) = 8</td>
</tr>
<tr>
<td></td>
<td>8</td>
<td>16</td>
<td>log(512) = 9</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>8</td>
<td></td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>2</td>
<td></td>
</tr>
</tbody>
</table>

\[ \log(32) = 5 \quad \log(64) = 6 \]

All logarithms in this course are base 2

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

\[ \log(\text{number of particles in the universe}) < 280 \]
It’s fast!

CLAIM:

MERGESORT requires at most $11n (\log(n) + 1)$ operations to sort $n$ numbers.

Much faster than InsertionSort for large $n$! (No matter how the algorithms are implemented). (And no matter what that constant “11” is).
Let’s prove the claim

- Later we’ll see more principled ways of analyzing divide-and-conquer algs.
- But for today let’s just wing it.

Focus on just one of these sub-problems

(Size 1)
How much work in this sub-problem?

\[ \frac{n}{2^t} \]

Time spent MERGE-ing the two subproblems

\[ \frac{n}{2^{t+1}} + \frac{n}{2^{t+1}} \]

Time spent within the two sub-problems
How much work in this sub-problem?

Let $k = n/2^t$...

Time spent MERGE-ing the two subproblems

Time spent within the two sub-problems
How long does it take to MERGE?

Code for the MERGE step is given in the Lecture2 notebook.
How long does it take to MERGE?

- Time to initialize an array of size $k$
- Plus the time to initialize three counters
- Plus the time to increment two of those counters $k/2$ times each
- Plus the time to compare two values at least $k$ times
- Plus the time to copy $k$ values from the existing array to the big array.
- Plus...

Let’s say no more than $11k$ operations.

There’s some justification for this number “11” in the lecture notes, but it’s really pretty arbitrary.

Code for the MERGE step is given in the Lecture2 notebook.
## Recursion tree

<table>
<thead>
<tr>
<th>Level</th>
<th># problems</th>
<th>Size of each problem</th>
<th>Amount of work at this level</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>n</td>
<td>11n</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td>n/2</td>
<td>11n</td>
</tr>
<tr>
<td>2</td>
<td>4</td>
<td>n/4</td>
<td>11n</td>
</tr>
<tr>
<td></td>
<td>...</td>
<td></td>
<td></td>
</tr>
<tr>
<td>t</td>
<td>2^t</td>
<td>n/2^t</td>
<td>11n</td>
</tr>
<tr>
<td></td>
<td>...</td>
<td></td>
<td></td>
</tr>
<tr>
<td>log(n)</td>
<td>n</td>
<td>1</td>
<td>11n</td>
</tr>
</tbody>
</table>

Amount of work at a level:
(number of problems) \times 11 \times (size of problem)

[explanation on board]

(Size 1)
Total runtime...

• $11n$ steps per level, at every level
• $\log(n) + 1$ levels

• $11n \ (\log(n) + 1)$ steps total

That was the claim!
A few reasons to be grumpy

• Sorting

    1  2  3  4  5  6  7  8

should take zero steps...

• What’s with this 11k bound?
  • You (Mary) made that number “11” up.
  • Different operations don’t take the same amount of time.
How we will deal with **grumpiness**

- Take a deep breath...
- Worst case analysis
- Asymptotic notation
The plan

• **Part I: Sorting Algorithms**
  • InsertionSort: does it work and is it fast?
  • MergeSort: does it work and is it fast?
  • **Skills:**
    • Analyzing correctness of iterative and recursive algorithms.
    • Analyzing running time of recursive algorithms (part A)

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

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

Pros: very strong guarantee
Cons: very strong guarantee
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.

How long does an operation take? Why are we being so sloppy about that “11”? 
Main idea:

Focus on how the runtime **scales** with n (the input size).
Asymptotic Analysis
How does the running time scale as $n$ gets large?

One algorithm is “faster” than another if its runtime scales better with the size of the input.

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).

$2^{1000000000000000} n$ is “better” than $n^2$ ?!?!
O(...) means an upper bound

• Let $T(n)$, $g(n)$ be functions of positive integers.
  • Think of $T(n)$ as being a runtime: positive and increasing in $n$.

• We say “$T(n)$ is $O(g(n))$” if $g(n)$ grows at least as fast as $T(n)$ as $n$ gets large.

• Formally,

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

pronounced “big-oh of ...” or sometimes “oh of ...”
Example

\[2n^2 + 10 = O(n^2)\]

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

\[ 2n^2 + 10 = O(n^2) \]

Formally:

- Choose \( c = 3 \)
- Choose \( n_0 = 4 \)
- Then:

\[ 0 \leq 2n^2 + 10 \leq 3 \cdot n^2 \]
same Example

\[ 2n^2 + 10 = O(n^2) \]

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

Formally:
- Choose \( c = 7 \)
- Choose \( n_0 = 2 \)
- Then:
  \[
  \forall n \geq 2, \\
  0 \leq 2n^2 + 10 \leq 7 \cdot n^2
  \]

There is not a “correct” choice of \( c \) and \( n_0 \)
Another example:

\[ n = O(n^2) \]

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

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

- Choose \( c = 1 \)
- Choose \( n_0 = 1 \)
- Then

\[ \forall n \geq 1, \]

\[ 0 \leq n \leq n^2 \]
\( \Omega(\ldots) \) means a lower bound

- We say “\( T(n) \) is \( \Omega(g(n)) \)” if \( g(n) \) grows at most as fast as \( T(n) \) as \( n \) gets large.

- Formally,

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

\[ n \log_2(n) = \Omega(3n) \]

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

- Choose \( c = 1/3 \)
- Choose \( n_0 = 3 \)
- Then

\[ \forall n \geq 3, \]
\[ 0 \leq \frac{3n}{3} \leq n \log_2(n) \]
Θ(...) means both!

- We say “T(n) is Θ(g(n))” if:

\[
T(n) = O(g(n)) \quad \text{-AND-} \quad T(n) = \Omega(g(n))
\]
Some more examples

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

*Need some caveat here...what is it?

(On the board if we have time... if not see the lecture notes!)
Take-away from examples

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

• To prove $T(n)$ is NOT $O(g(n))$, one way is **proof by contradiction**:
  • Suppose (to get a contradiction) 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.
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 \( O(2^n) \)
- \( \log(n) = \Omega(\ln(n)) \)
- \( \log(n) = \Theta(2^{\log\log(n)}) \)

Work through any of these that we don’t have time to go through in class!

remember that \( \log = \log_2 \) in this class.
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:
• \( F(n) = O(2^n) \)
• \( F(n) = \Omega(2^n) \)

Ollie the Over-achieving Ostrich
What have we learned?
Asymptotic Notation

• This makes both Plucky and Lucky happy.
  • **Plucky the Pedantic Penguin** is happy because there is a precise definition.
  • **Lucky the Lackadaisical Lemur** is happy because we don’t have to pay close attention to all those pesky constant factors like “11”.

• 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 $n \geq n_0 = 2^{100000000}$.
The plan

- **Part I: Sorting Algorithms**
  - InsertionSort: does it work and is it fast?
  - MergeSort: does it work and is it fast?
  - **Skills:**
    - Analyzing correctness of iterative and recursive algorithms.
    - Analyzing running time of recursive algorithms (part A)

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

Wrap-Up
Recap

• **InsertionSort** runs in time $O(n^2)$

• **MergeSort** is a divide-and-conquer algorithm that runs in time $O(n \log(n))$

• How do we show an algorithm is correct?
  • Today, we did it by induction

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

• A more systematic approach to analyzing the runtime of recursive algorithms.

Before next time

• Pre-Lecture Exercise:
  • A few recurrence relations (see website)