Lecture 2

Asymptotic Notation,
Worst-Case Analysis, and MergeSort
Announcements

• HW0 Due Friday!
• HW1 Out now! Due Wednesday!
• Homework Party!
  • Thursday 6-8pm, Gates 403 (Fujitsu)
  • There will be pizza! 🍕🍕🍕
• Sections!
  • Thursday and Friday!
  • One section is recorded!
• Fill out form about possible experiment in class! By THURSDAY!
  • See Ed announcement
• Please (continue to) send OAE letters and exam conflicts to cs161-spr2223-staff@lists.stanford.edu
  • Let us know by Wednesday April 12!
  • For exam conflicts you can also just fill out a Google form (see Ed post)

Midterm: Thursday May 4, 6-9pm
Final: Monday June 12, 3:30-6:30pm
Homework!

• HW will be done in groups (of size up to 3)
  • Individually is fine too if you prefer

• Homework comes in two parts:
  • Exercises:
    • More straightforward.
    • Try to do them on your own.
  • Problems:
    • Less straightforward, good opportunity for group brainstorming!

• See the website for guidelines on homework:
  • Collaboration + Late Day policy (in the “Policies” tab)
  • Example Homework (in the “Resources” tab)
  • LaTeX help (in the “Resources” tab)
Resources!

• If HW0 was tough:
  • See Ed announcement for links to a bunch of background materials (thanks to Ian Tullis!) if you want to brush up on logarithms, induction, probability, etc.
  • The textbook has nice appendices about pf by induction and probability.
  • Go to section! Go to OH!

• As you get into HW1:
  • We’ve posted a “problem-solving guide” to give you some tips about how to do the “creative problem-solving” part of the course!
  • Homework party! 🍕🍕🍕
Promise

• At least one problem on each exam will be similar to a HW problem!
Did anyone find an apple pencil?
End of announcements!
Last time

Philosophy

- Algorithms are awesome!
- Our motivating questions:
  - Does it work?
  - Is it fast?
  - Can I do better?

Technical content

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

• We are going to ask:
  • Does it work?
  • Is it fast?

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

SortingHatSort not discussed
The Plan

• Sorting!
• Worst-case analysis
  • InsertionSort: Does it work?
• Asymptotic Analysis
  • InsertionSort: Is it fast?
• MergeSort
  • Does it work?
  • Is it fast?
Sorting

• Important primitive
• For today, we’ll pretend all elements are distinct.

Length of the list is n
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):
    for x in 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
```
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):
    for x in 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
```
InsertionSort

equation

example

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

Then move A[3]:

Then move A[2]:

Then move A[4]:

Then we are done!
Insertion Sort

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

What does that mean???
The Plan

• InsertionSort recap
• Worst-case Analysis
  • Back to InsertionSort: Does it work?
• Asymptotic Analysis
  • Back to InsertionSort: Is it fast?
• MergeSort
  • Does it work?
  • Is it fast?
Claim: InsertionSort “works”

• “Proof:” It just worked in this example:

```
6 4 3 8 5
6 4 3 8 5
4 6 3 8 5
4 6 3 8 5
3 4 6 8 5
3 4 6 8 5
3 4 6 8 5
3 4 6 8 5
3 4 5 6 8
Sorted!
```
Claim: InsertionSort “works”

• “Proof:” I did it on a bunch of random lists and it always worked:

```python
A = [1,2,3,4,5,6,7,8,9,10]
for trial in range(100):
    shuffle(A)
    InsertionSort(A)
    if is_sorted(A):
        print('YES IT IS SORTED!')
```
What does it mean to “work”?

• Is it enough to be correct on only one input?
• Is it enough to be correct on most inputs?

• In this class, we will use worst-case analysis:
  • An algorithm must be correct on all possible inputs.
  • The running time of an algorithm is the worst possible running time over all inputs.
Worst-case analysis

Think of it like a game:

Here is my algorithm!

Algorithm:
  Do the thing
  Do the stuff
  Return the answer

Here is an input!
(Which I designed to be terrible for your algorithm!)

Worst-case analysis guarantee: Algorithm should work (and be fast) on that worst-case input.

• Pros: very strong guarantee
• Cons: very strong guarantee
Insertion Sort

1. Does it work?
   - Okay, so it’s pretty obvious that it works.

2. Is it fast?
   - 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.

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

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

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

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

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

So correctly inserting 8 into the list [3,4,6] means that [3,4,6,8] becomes 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!
This sounds like a job for...

Proof By Induction!
There is a handout with details!

• See website!

2 Correctness of InsertionSort

Once you figure out what InsertionSort is doing (see the slides/lecture video for the intuition on this), you may think that it’s “obviously” correct. However, if you didn’t know what it was doing and just got the above code, maybe this wouldn’t be so obvious. Additionally, for algorithms that we’ll study in the future, it won’t always be obvious that it works, and so we’ll have to prove it. So in this handout we’ll carefully go through a proof that InsertionSort is correct.

We’ll do the proof by maintaining a loop invariant, in this case that after iteration $i$, then $A[:i+1]$ is sorted. This is obviously true after iteration 0 (aka, before the algorithm begins), because the one-element list $A[:1]$ is definitely sorted. Then we’ll show that for any $k$ with $0 < k < n$, if this loop invariant holds for $k-1$, then it holds for $k$. That is, if it is true that $A[:k]$ is sorted after the $k-1$’st iteration, then it is true that $A[:k+1]$ is sorted after the $k$’th iteration. At the end of the day, we’ll conclude that $A[:n]$ (aka, the whole thing) is sorted after the $n-1$’st iteration, and we’ll be done.

Formally, we will proceed by induction.

• **Inductive hypothesis.** After iteration $i$ of the outer loop, $A[:i+1]$ is sorted.

• **Base case.** After iteration 0 of the outer loop (aka, before the algorithm begins), the list $A[:1]$ contains only one element, and this is sorted.

• **Inductive step.** Let $k$ be an integer so that $0 < k < n$. Suppose that the inductive hypothesis holds for $k-1$, so $A[:k]$ is sorted after the $k-1$’st iteration. We want to show that $A[:k+1]$ is sorted after the $k$’th iteration.

Suppose that $j^*$ is the largest integer in $\{0, \ldots, k-1\}$ such that $A[j^*] < A[k]$. Then the effect of the inner loop is to turn $[A[0], A[1], \ldots, A[j^*], \ldots, A[k-1], A[k]]$ into $[A[0], A[1], \ldots, A[j^*], A[k], A[j^*+1], \ldots, A[k-1]]$. 

Outline of a proof by induction

Let A be a list of length n

• Inductive Hypothesis:
  • \( A[{:i+1}] \) is sorted at the end of the \( i^{th} \) iteration (of the outer loop).

• Base case (i=0):
  • \( A[{:1}] \) is sorted at the end of the 0’th iteration. ✓

• Inductive step:
  • For any \( 0 < k < n \), if the inductive hypothesis holds for \( i=k-1 \), then it holds for \( i=k \).
  • Aka, if \( A[{:k}] \) is sorted at step \( k-1 \), then \( A[{:k+1}] \) is sorted at step \( k \)

• Conclusion:
  • The inductive hypothesis holds for \( i = 0, 1, ..., n-1 \).
  • In particular, it holds for \( i=n-1 \).
  • \( A[{:n}] \) is sorted at the end of the \( n-1^{st} \) iteration
  • Aka, A is sorted at the end of the algorithm! ✓

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.

This was iteration \( i=2 \).
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:
  • GO TO SECTION
  • GO TO SECTION
  • Lecture 2 Handout
  • Book (appendix A)
  • Ian Tullis’s notes
  • Office Hours

Make sure you really understand the argument on the previous slide! Check out the handout for a more formal write-up, and go to section for an overview of what we are looking for in proofs by induction.
What have we learned?

• In this class we will use worst-case analysis:
  • We assume that a “bad guy” comes up with a worst-case input for our algorithm, and we measure performance on that worst-case input.

• With this definition, InsertionSort “works”
  • Proof by induction!
The Plan

• InsertionSort recap
• Worst-case Analysis
  • Back to InsertionSort: Does it work?
• Asymptotic Analysis
  • Back to InsertionSort: Is it fast?
• MergeSort
  • Does it work?
  • Is it fast?
How fast is InsertionSort?

• This fast:
Issues with this answer?

• The “same” algorithm can be slower or faster depending on the implementations.
• It can also be slower or faster depending on the hardware that we run it on.
• It might also be slower or faster depending on the inputs we use.
• What if n=2000?

With this answer, “running time” isn’t even well-defined!
How fast is InsertionSort?

• Let’s count the number of operations!

```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
```

By my count*...
• $2n^2 - n - 1$ variable assignments
• $2n^2 - n - 1$ increments/decrements
• $2n^2 - 4n + 1$ comparisons
• ...

*Do not pay attention to these formulas, they do not matter. Also not valid for bug bounty points.
Issues with this answer?

• It’s very tedious!

• Might be slightly different for slightly different implementations.

• In order to use this to understand running time, I need to know how long each operation takes, plus a whole bunch of other stuff...

```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
```

Counting individual operations is a lot of work and doesn’t seem very helpful!

Lucky the lackadaisical lemur
In this class we will use...

• **Big-Oh notation!**

• Gives us a meaningful way to talk about the running time of an algorithm, independent of programming language, computing platform, etc., without having to count all the operations.
Main idea:

Focus on how the runtime **scales** with n (the input size).

Some examples...

<table>
<thead>
<tr>
<th>Number of operations</th>
<th>Asymptotic Running Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\frac{1}{10} \cdot n^2 + 100$</td>
<td>$O(n^2)$</td>
</tr>
<tr>
<td>$0.063 \cdot n^2 - 0.5 n + 12.7$</td>
<td>$O(n^2)$</td>
</tr>
<tr>
<td>$100 \cdot n^{1.5} - 10^{100000} \sqrt{n}$</td>
<td>$O(n^{1.5})$</td>
</tr>
<tr>
<td>$11 \cdot n \log(n) + 1$</td>
<td>$O(n \log(n))$</td>
</tr>
</tbody>
</table>

(Only pay attention to the largest function of n that appears.)

We say this algorithm is "asymptotically faster" than the others.
Why is this a good idea?

• Suppose the running time of an algorithm is:

\[ T(n) = 10n^2 + 3n + 7 \text{ ms} \]

This constant factor of 10 depends a lot on my computing platform...

These lower-order terms don’t really matter as \( n \) gets large.

We’re just left with the \( n^2 \) term! That’s what’s meaningful.
Pros and Cons of Asymptotic Analysis

Pros:

• Abstracts away from hardware- and language-specific issues.
• Makes algorithm analysis much more tractable.
• Allows us to meaningfully compare how algorithms will perform on large inputs.

Cons:

• Only makes sense if \( n \) is large (compared to the constant factors).

\[ 10000000000 \ n \text{ is “better” than } n^2 \ ?!?! \]
Informal definition for $O(...)$

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

- We say “$T(n)$ is $O(g(n))$” if:
  
  for large enough $n$,
  
  $T(n)$ is at most some constant multiple of $g(n)$.

Here, “constant” means “some number that doesn’t depend on $n$.”

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

$2n^2 + 10 = O(n^2)$

for large enough $n$, $T(n)$ is at most some constant multiple of $g(n)$. 

$T(n) = 2n^2 + 10$

$g(n) = n^2$
Example

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

for large enough \( n \), \( T(n) \) is at most some constant multiple of \( g(n) \).
Example

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

for large enough \( n \), \( T(n) \) is at most some constant multiple of \( g(n) \).
Formal definition of $O(...)$

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

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

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

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

$2n^2 + 10 = O(n^2)$

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

$T(n) = 2n^2 + 10$

$g(n) = n^2$

$3g(n) = 3n^2$

$3 \cdot g(n) = 3n^2$

(c=3)
Example

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

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

Formally:

- Choose \( c = 3 \)
- Choose \( n_0 = 4 \)
- Then:
  \[ \forall n \geq 4, \quad 2n^2 + 10 \leq 3 \cdot n^2 \]
Same example

$2n^2 + 10 = O(n^2)$

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

There is not a “correct” choice of $c$ and $n_0$
O(...) is an upper bound:
\[ n = O(n^2) \]

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

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

\[ \forall n \geq 1, \quad n \leq n^2 \]
Ω(...)

- We say “$T(n)$ is $\Omega(g(n))$” if, for large enough $n$, $T(n)$ is at least as big as a constant multiple of $g(n)$.

- Formally,

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

$\iff$

$$\exists c, n_0 > 0 \ s.t. \ \forall n \geq n_0, \ \ c \cdot g(n) \leq T(n)$$

Switched these!!
Example

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

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

- Choose \( c = 1/3 \)
- Choose \( n_0 = 2 \)
- Then

\[ \forall n \geq 2, \]
\[ \frac{3n}{3} \leq n \log_2(n) \]
\(\Theta(...)\) means both!

- We say "\(T(n)\) is \(\Theta(g(n))\)" iff both:

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

and

\[
T(n) = \Omega(g(n))
\]
Non-Example: \( n^2 \) is not \( O(n) \)

- Proof by contradiction:
- Suppose that \( n^2 = O(n) \).
- Then there is some positive \( c \) and \( n_0 \) so that:
  \[
  \forall n \geq n_0, \quad n^2 \leq c \cdot n
  \]
- Divide both sides by \( n \):
  \[
  \forall n \geq n_0, \quad n \leq c
  \]
- That’s not true!!! What about, say, \( n_0 + c + 1 \)?
  - Then \( n \geq n_0 \), but \( n > c \)
- Contradiction!
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 be lying to you by deriving a contradiction.
Another example: polynomials

• Say \( p(n) = a_k n^k + a_{k-1} n^{k-1} + \cdots + a_1 n + a_0 \) is a polynomial of degree \( k \geq 1 \).

• Then:
  1. \( p(n) = O(n^k) \)
  2. \( p(n) \) is **not** \( O(n^{k-1}) \)

• See the book (AI Section 2.3.2) for a proof.

Try to prove it yourself first!
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** $O(2^n)$
• $\log_2(n) = \Omega(\ln(n))$
• $\log_2(n) = \Theta(2^{\log\log(n)})$

Work through these on your own! Also look at the examples in the reading!
Some brainteasers

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

• Are there non-decreasing functions $f, g$ so that the above is true?
Recap: 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.

• 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 $n \geq n_0 = 2^{100000000}$. 
Back Insertion Sort

1. Does it work?
2. Is it fast?
Insertion Sort: running time

• Operation count was:
  • $2n^2 - n - 1$ variable assignments
  • $2n^2 - n - 1$ increments/decrements
  • $2n^2 - 4n + 1$ comparisons
  • ...

• The running time is $O(n^2)$

Go back to the pseudocode and convince yourself of this!
Insertion Sort: running time

As you get more used to this, you won’t have to count up operations anymore. For example, just looking at the pseudocode below, you might think…

```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

“There’s \( O(1) \) stuff going on inside the inner loop, so each time the inner loop runs, that’s \( O(n) \) work. Then the inner loop is executed \( O(n) \) times by the outer loop, so that’s \( O(n^2) \).”
What have we learned?

InsertionSort is an algorithm that correctly sorts an arbitrary n-element array in time $O(n^2)$.

Can we do better?
The Plan

• InsertionSort recap
• Worst-case analysis
  • Back to InsertionSort: Does it work?
• Asymptotic Analysis
  • Back to InsertionSort: Is it fast?
• MergeSort
  • Does it work?
  • Is it fast?
Can we do better?

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

```
Big problem

Smaller problem

Yet smaller problem

Smaller problem

Yet smaller problem

Yet smaller problem

Yet smaller problem

Yet smaller problem

Yet smaller problem

Recurse!

Recurse!
```
MergeSort

How would you do this in-place?

Recursive magic!

MERGE!

Code for the MERGE step is given in the Lecture2 IPython notebook, or the textbook.

Ollie the over-achieving Ostrich
MergeSort Pseudocode

MERGESORT(A):

• n = length(A)

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

• L = MERGESORT(A[0 : n/2])  
  Sort the left half

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

• return MERGE(L, R)  
  Merge the two halves

See Lecture 2 IPython notebook for MergeSort Python Code.
Two questions

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

Empirically:
1. Seems to work.
2. Seems fast.

IPython notebook says...

![Graph showing time vs. n for different sorting algorithms]
It works

• Yet another job for...

Proof By Induction!

Work this out! There’s a skipped slide with an outline to help you get started.
Outline!

• **Inductive hypothesis (IH):**
  “In every recursive call on an array of length at most i, MERGESORT returns a sorted array.”

• **Base case (i=1):** a 1-element array is always sorted, so IH holds for i=1.

• **Inductive step:** Need to show: if IH holds for all 0<i<k, then it holds for i=k.
  Aka, need to show that if L and R are sorted, then MERGE(L,R) is sorted.

• **Conclusion:** The IH holds for i=1,2,...,n, and in particular for n.
  Aka, In the top recursive call, MERGESORT returns a sorted array!

• **MERGESORT** (A):
  • n = length(A)
  • if n ≤ 1:
    • return A
  • L = MERGESORT(A[0 : n/2])
  • R = MERGESORT(A[n/2 : n])
  • return MERGE(L,R)

Fill in the inductive step!
HINT: You will need to prove that the MERGE algorithm is correct, for which you may need...another proof by induction!

ASSUME THAT n IS A POWER OF 2 FOR CONVENIENCE.
It’s fast

CLAIM:

MergeSort runs in time $O(n \log(n))$

• Proof coming soon.
• But first, how does this compare to InsertionSort?
  • Recall InsertionSort ran in time $O(n^2)$.

Assume that $n$ is a power of 2 for convenience.
$O(n \log(n))$ vs. $O(n^2)$?
Quick log refresher

• **Def:** \( \log(n) \) is the number so that \( 2^{\log(n)} = n \).

• **Intuition:** \( \log(n) \) is how many times you need to divide \( n \) by 2 in order to get down to 1.

\[
32, 16, 8, 4, 2, 1 \quad \Rightarrow \quad \log(32) = 5
\]

Halve 5 times

\[
64, 32, 16, 8, 4, 2, 1 \quad \Rightarrow \quad \log(64) = 6
\]

Halve 6 times

\[
\log(128) = 7 \\
\log(256) = 8 \\
\log(512) = 9 \\
\ldots \\
\log(\# \text{ particles in the universe}) < 280
\]

• **\( \log(n) \) grows very slowly!**
$O(n \log n)$ vs. $O(n^2)$?

- $\log(n)$ grows much more slowly than $n$
- $n \log(n)$ grows much more slowly than $n^2$

Punchline: A running time of $O(n \log n)$ is a lot better than $O(n^2)$!
Now let’s prove the claim

CLAIM:

MergeSort runs in time $O(n \log(n))$

Assume that $n$ is a power of 2 for convenience.
Let’s prove the claim

Focus on just one of these sub-problems.

2^t subproblems at level t.

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

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

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

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

Time spent MERGE-ing the two subproblems

\[ + \]

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

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

Time spent MERGE-ing the two subproblems

$+\quad$ 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?

Question: in big-Oh notation, how long does it take to run MERGE on two lists of size $k/2$?

Answer: It takes time $O(k)$, since we just walk across the list once.

Take-away: There are $O(k)$ operations done at this node. (Not including work at recursive calls.)

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

There are $O(k)$ operations done at this node.
Recursion tree

Size n

How many operations are done at this level of the tree? (Just MERGE-ing subproblems).

How about at this level of the tree? (just MERGE-ing, between both n/2-sized problems)

This level?

This level?

There are $O(k)$ operations done at this node.

Think, Pair, Share!

(Size 1)
## Recursion Tree

### Diagram

- Size $n$
- $n/2$
- $n/2$
- $n/4$
- $n/4$
- $n/4$
- $n/4$
- $n/2^t$
- $n/2^t$
- $n/2^t$
- $n/2^t$
- $n/2^t$
- $n/2^t$
- (Size 1)

### Table

<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>$O(n)$</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td>$n/2$</td>
<td>$O(n)$</td>
</tr>
<tr>
<td>2</td>
<td>4</td>
<td>$n/4$</td>
<td>$O(n)$</td>
</tr>
<tr>
<td>$t$</td>
<td>$2^t$</td>
<td>$n/2^t$</td>
<td>$O(n)$</td>
</tr>
<tr>
<td>log($n$)</td>
<td>$n$</td>
<td>1</td>
<td>$O(n)$</td>
</tr>
</tbody>
</table>

**Explanation for this table done on the board!**
Total runtime...

- $O(n)$ steps per level, at every level
- $\log(n) + 1$ levels

- $O(n \log(n))$ total!

That was the claim!
What have we learned?

• MergeSort correctly sorts a list of $n$ integers in time $O(n \log(n))$.
• That’s (asymptotically) better than InsertionSort!
The Plan

• InsertionSort recap
• Worst-case analysis
  • Back to InsertionSort: Does it work?
• Asymptotic Analysis
  • Back to InsertionSort: Is it fast?
• MergeSort
  • Does it work?
  • Is it fast?

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

• How do we analyze the running time of a recursive algorithm?
  • One way is to draw a recursion tree.
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)

• Do HW0 (due Friday!)
• Fill out form about study (by tomorrow!)