Analysis of Algorithms & Computational Complexity

Slides based on materials provided by Mary Wootters (Stanford University)
Today

• Sorting: InsertionSort vs MergeSort

• Analyzing the correctness of algorithms

• Analyzing the running time of algorithms
  • 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)

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

5 | 2 | 4 | 6 | 1 | 3
2 | 5 | 4 | 6 | 1 | 3
2 | 4 | 5 | 6 | 1 | 3
2 | 4 | 5 | 6 | 1 | 3
1 | 2 | 4 | 5 | 6 | 1 | 3
1 | 2 | 3 | 4 | 5 | 6 | 1 | 3
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 \((\text{len}(A) + 1)\)’st iteration (aka, at the end of the algorithm), the first \text{len}(A) items are sorted. ✓
Insertion sort: running time

Algorithm 1: INSERTIONSORT(A)

for \( i = 2 \rightarrow \text{length}(A) \) do

\[
\begin{align*}
\text{key} & \leftarrow A[i]; \\
\text{j} & \leftarrow i - 1; \\
\text{while } j > 0 \text{ and } A[j] > \text{key} \text{ do} & \\
& A[j + 1] \leftarrow A[j]; \\
& j \leftarrow j - 1; \\
& A[j + 1] \leftarrow \text{key};
\end{align*}
\]

Running time is about \( n^2 \)
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 previous courses:

![Diagram showing a divide and conquer approach with recursive calls leading to yet smaller problems.](image-url)
MergeSort

[6 4 3 8 1 5 2 7]

6 4 3 8

Recursive magic!

3 4 6 8

Recursive magic!

1 5 2 7

MERGE!

1 2 3 4 5 6 7 8
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
## Mergesort: Example

<table>
<thead>
<tr>
<th>Steps</th>
<th>a[]</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0</td>
</tr>
</tbody>
</table>
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 \( \text{MERGE}(L,R) \) 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 \( \text{MERGE}(L,R) \)
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) \text{ vs } n^2

- \log(n) : \text{ how many times do you need to divide } n \text{ 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></td>
</tr>
<tr>
<td>8</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
</tr>
</tbody>
</table>

32 \rightarrow 16 \rightarrow 8 \rightarrow 4 \rightarrow 2 \rightarrow 1

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

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

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

\log(\text{number of particles in the universe}) < 280

All logarithms in this course are base 2
**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

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

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

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

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

- 3 variable assignments (counters ← 1)
- n comparisons
- n more assignments
- 2n counter increments

So that’s

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

or \( 4n + 2 \) operations

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.
### Recursion Tree

#### Diagram:
- Size $n$
- $n/2$
- $n/2$
- $n/4$
- $n/4$
- $n/4$
- $n/4$

#### Table:

<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>

**Amount of work at a level:**
- $(\text{number of problems}) \times 6 \times (\text{size of problem})$
- (explanation on board)
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 $T(\text{MERGE}) < 6n$?

  • $2 + 4n \leq 6n$ is a loose bound.
  • Different operations don’t take the same amount of time.
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.
n log(n) vs n^2

<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>

Time in milliseconds

Graph showing the comparison of n log(n) vs n^2.
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>

Time in milliseconds

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).

$2^{100000000000000} n$ is “better” than $n^2$ ?!?!
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,

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

(formal proof on board)
Example 2

• \( g(n) = 2, \ f(n) = 1 \).

• \( g(n) = O(f(n)) \) (and also \( f(n) = O(g(n)) \))

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

\( T(n) = 2.1 \cdot f(n) \) with \( n_0 = 1 \) and \( c = 2.1 \).
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,

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

Switched these!!
Parsing that...

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

- We say “T(n) is Θ(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 $O(2^n)$
• $n \log(n) = \Omega(n)$
• $\log(n) = \Theta(2^{\log\log(n)})$

(on board if time – otherwise try them yourself!)
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 $n \geq n_0 = 2^{10000000}$. 