Putting the “Science” in Computer Science

What makes for a good program, and how can we measure / evaluate programs for “goodness”? Write as many definitions of “good” as you can, and describe how you would measure each one.
Given a computational problem
Is there a solution?
What is it?
How good is it?
Is it efficient?
Data: which algorithm is best?

Lower is better
Data: which algorithm is best?

Lower is better
Data: which algorithm is best?

Lower is better
Data: which algorithm is best?

Lower is better
Data: which algorithm is best?

Lower is better
Data: which algorithm is best?

Lower is better
Data: which algorithm is best?

Lower is better
Data: which algorithm is best?

Lower is better
Data: which algorithm is best?

Lower is better
Data: which algorithm is best?

Lower is better
Interpreting empirical data

Key take-away: it’s messy and incomplete!

We can measure

- a particular **algorithm**
- written in a particular **language**
- as a particular **program**
- compiled using a particular version of a particular **compiler**
- with particular **settings** (e.g., enabling / disabling optimizations)
- running on a particular **data set**, of a particular **size**
- on a particular **computer**
- with particular **resources** (CPUs, memory, hard drive, …)
- under a particular version of a particular **operating system**
- in a particular **environment**
  
  e.g., with other programs running in the background
Interpreting a theoretical model

A theory abstracts away certain details.

**cost metric:**
- corresponds to one “step”
- highlights the essence of the work
e.g., multiplications, comparisons, function calls…
- serves as a proxy for an empirical measurement

Instead of measuring time, we count steps.
e.g., “This algorithm costs $n^2$ multiplications.”
good data + good theory

= 

good science

we can make predictions and
we can communicate with other scientists
Decidability
- e.g., Can it be solved at all?

Complexity Class
- e.g., Can it be solved in polynomial time?

Asymptotic Analysis
- e.g., $O(n)$ time, where $n$ is list size

Exact Theory
- e.g., $7n + 2$ multiplications, where $n$ is list size

Empirical Data
- e.g., This run took 17.3 seconds on this data.
Asymptotic Analysis (Big O)
Asymptotic analysis

We’re always answering the same question:

**How does the cost `scale` (when we try larger and larger inputs)?**

**Not:**
- Exactly how many steps will it execute?
- How many seconds will it take?
- How many megabytes of memory will it need?
A reasonable upper bound on (an abstraction of) a problem’s difficulty or a solution’s performance, for reasonably large input sizes.
In the limit (for VERY LARGE inputs)

The running time is bounded regardless of the input size. \( O(1) \)

An input twice as big takes no more than twice as long. \( O(n) \)

An input twice as big takes no more than four times as long. \( O(n^2) \)

An input one bigger takes no more than twice as long. \( O(2^n) \)
If We Only Care About Scalability...

What are the consequences?

Constant factors can be ignored.

\[ n \quad \text{and} \quad 6n \quad \text{and} \quad 200n \] scale identically ("linearly"")

Small summands can be ignored.

\[ n^2 \quad \text{and} \quad n^2 + n + 999999 \] are indistinguishable when \( n \) is huge.
Grouping Algorithms by Scalability

**O(1)**
- takes 6 steps
- takes 1 (big) step
- no more than 4000 steps
- somewhere between 2 and 47 steps, depending on the input

**O(n)**
- takes $100n + 3$ steps
- takes $\frac{n}{20} + 10,000,000$ steps
- anywhere between 3 and 68 steps per item, for $n$ items.

**O(n^2)**
- takes $2n^2 + 100n + 3$ steps
- takes $\frac{n^2}{17}$ steps
- somewhere between 1 and 40 steps per item, for $n^2$ items
- anywhere between 1 and $7n$ steps per item, for $n$ items.
How hard is the problem?

- $O(n^n)$
- $O(n!)$
- $O(2^n)$
- $O(n^3)$
- $O(n^2)$
- $O(n \log(n))$
- $O(n)$
- $O(\sqrt{n})$
- $O(\log(n))$
- $O(1)$

**Intractable problems** (exponential)

**Tractable problems** (polynomial)

**No problem!**
logs aren’t scary!
They’re our friends.

\[ \log_2 N = p \iff 2^p = N \]

log is the inverse of exponentiation.

How many times can I cut \( N \) in half?

Can I avoid looking at all the input?!

\[
\begin{align*}
\log_2(1) &= 0 \quad // 2^0 = 1 \\
\log_2(2) &= 1 \quad // 2^1 = 2 \\
\log_2(3) &\approx 1.58 \\
\log_2(4) &= 2 \quad // 2^2 = 4 \\
\log_2(5) &\approx 2.32 \\
\log_2(6) &\approx 2.58 \\
\log_2(7) &\approx 2.81 \\
\log_2(8) &= 3 \quad // 2^3 = 8
\end{align*}
\]
How hard are these problems?

<table>
<thead>
<tr>
<th>cost metric</th>
<th>cost</th>
</tr>
</thead>
<tbody>
<tr>
<td>double</td>
<td>multiplications</td>
</tr>
<tr>
<td>sum</td>
<td>additions</td>
</tr>
<tr>
<td>half-count</td>
<td>divisions</td>
</tr>
</tbody>
</table>
How hard are these problems?

<table>
<thead>
<tr>
<th>cost metric</th>
<th>cost</th>
</tr>
</thead>
<tbody>
<tr>
<td>double</td>
<td></td>
</tr>
<tr>
<td>multiplications</td>
<td>O(1)</td>
</tr>
<tr>
<td>sum</td>
<td></td>
</tr>
<tr>
<td>additions</td>
<td>O(n)</td>
</tr>
<tr>
<td>half-count</td>
<td></td>
</tr>
<tr>
<td>divisions</td>
<td>O(log n)</td>
</tr>
</tbody>
</table>
What’s the cost, $T$, for each function?

<table>
<thead>
<tr>
<th>function</th>
<th>double multiplications</th>
<th>sum additions</th>
<th>half-count divisions</th>
</tr>
</thead>
<tbody>
<tr>
<td>(define (double n) (* n 2))</td>
<td>$T(0)$</td>
<td>$T(1)$</td>
<td>$T(2)$</td>
</tr>
<tr>
<td>(define (sum n) (if (= n 0) 0 (+ n (sum (- n 1)))))</td>
<td>$T(3)$</td>
<td>$T(4)$</td>
<td>...</td>
</tr>
<tr>
<td>(define (half-count n) (if (= n 1) 0 (+ 1 (half-count (quotient n 2)))))</td>
<td>$T(n)$</td>
<td>$T(n)$</td>
<td>$n/a$</td>
</tr>
</tbody>
</table>
What’s the cost, T, for each function?

<table>
<thead>
<tr>
<th></th>
<th>double multiplications</th>
<th>sum additions</th>
<th>half-count divisions</th>
</tr>
</thead>
<tbody>
<tr>
<td>T(0)</td>
<td>1</td>
<td>0</td>
<td>n/a</td>
</tr>
<tr>
<td>T(1)</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>T(2)</td>
<td>1</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>T(3)</td>
<td>1</td>
<td>3</td>
<td>1</td>
</tr>
<tr>
<td>T(4)</td>
<td>1</td>
<td>4</td>
<td>2</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>T(n)</td>
<td>1</td>
<td>n</td>
<td>⌊log₂ n⌋</td>
</tr>
</tbody>
</table>

(define (double n)
  (* n 2))

(define (sum n)
  (if (= n 0)
    0
    (+ n (sum (- n 1)))))

(define (half-count n)
  (if (= n 1)
    0
    (+ 1 (half-count (quotient n 2))))))
Recurrence Relations
(translating code to math)
Translating recursion to recurrence relations

For a given cost metric: additions

1. Translate the base case(s), using specific input sizes
   How many steps does this base case take?

2. Translate the recursive case(s), using input size N
   Define $T(N)$ in terms of smaller cost

\[
\text{(define (sum n)} \quad \text{base case} \quad \rightarrow \quad T(0) = \begin{cases} 
0 & \text{if } n = 0 \\
sum \cdot n (\text{- n 1})) & \text{if } n > 0
\end{cases}
\]

\[
\text{recursive case} \quad \rightarrow \quad T(N) = \begin{cases} 
0 & \text{if } N = 0 \\
3 + T(N-1) & \text{if } N > 0
\end{cases}
\]

For a given cost metric: additions
Translating recursion to recurrence relations

For a given cost metric: additions

1. Translate the base case(s), using specific input sizes
   How many steps does this base case take?

2. Translate the recursive case(s), using input size N
   Define $T(N)$ in terms of smaller cost

\[
T(N) = 1 + T(N-1)
\]

\[
T(N) = 1 + 1 + T(N-2)
\]

\[
T(N) = 1 + 1 + 1 + T(N-3)
\]

\[
\vdots
\]

\[
T(N) = 1 + 1 + 1 + \ldots + 1 + T(N-N)
\]

\[
T(0) = 0
\]

\[
T(N) = 1 + T(N-1)
\]

\[
T(N) = 1*1 + T(N-1)
\]

\[
T(N) = 2*1 + T(N-2)
\]

\[
T(N) = 3*1 + T(N-3)
\]

\[
\vdots
\]

\[
T(N) = N*1 + T(N-N) = N \in O(N)
\]
Translating recursion to recurrence relations

For a given cost metric: arithmetic operations and comparisons

1. Translate the base case(s), using specific input sizes
   How many steps does this base case take?

2. Translate the recursive case(s), using input size N
   Define T(N) in terms of smaller cost

\[
\begin{align*}
T(0) &= 1 \\
T(N) &= 2 + T(N-1)
\end{align*}
\]

For a given cost metric:
- arithmetic operations
- comparisons

\[(\text{define} \ (\text{sum} \ n))\]

\[(\text{if} \ (= \ n \ 0))\]

\[0\]

\[(+ \ n \ (\text{sum} \ (- \ n \ 1))))\]
Translating recursion to recurrence relations

For a given cost metric: arithmetic operations and comparisons

1. Translate the base case(s), using specific input sizes
   How many steps does this base case take?

2. Translate the recursive case(s), using input size N
   Define \( T(N) \) in terms of smaller cost

\[
\text{(define} \ (\text{sum} \ n) \\
\text{(if} \ (= \ n \ 0) \\
\quad 0 \\
\quad (+ \ n \ (\text{sum} \ (- \ n \ 1))))
\]

\[
\begin{align*}
T(0) &= 1 \\
T(N) &= 3 + T(N-1) \\
&= 2*2 + T(N-2) \\
&= 3*2 + T(N-3) \\
&= \ldots \\
&= N*3 + T(N-N) = 3N +1 \in O(N)
\end{align*}
\]
Translating recursion to recurrence relations

For a given cost metric: divisions

1. Translate the base case(s), using specific input sizes
   How many steps does this base case take?

2. Translate the recursive case(s), using input size N
   Define \( T(N) \) in terms of smaller cost

\[
\text{(define (half-count n)} \\
\text{  (if (= n 1)} \\
\text{    0)} \\
\text{  (+ 1 (half-count (quotient n 2)))))}
\]

\[
\begin{align*}
T(1) &= 1 \\
T(N) &= 3 + T(N-1)
\end{align*}
\]
Translating recursion to recurrence relations

For a given cost metric: divisions

1. Translate the base case(s), using specific input sizes
   How many steps does this base case take?

2. Translate the recursive case(s), using input size N
   Define $T(N)$ in terms of smaller cost

   $$(\text{define (half-count n)})$$
   $$(\text{(if (= n 1)})$$
   $$0$$
   $$(+ 1 (\text{half-count (quotient n 2)})))$$

   $$T(N) = 1 + T(N/2)$$
   $$= 1 + 1 + T(N/4)$$
   $$= 1 + 1 + 1 + T(N/8)$$
   $$\ldots$$
   $$= 1 + 1 + 1 + \ldots 1 + T(N/N)$$

   closed form
   $$T(1) = 0$$

   asymptotic form
   $$T(N) = 1 + T(N/2)$$
   $$= 2 + T(N/4)$$
   $$= 3 + T(N/8)$$
   $$\ldots$$
   $$= \log_2 N + T(N/N) = \log_2 N \in O(\log N)$$
Three problems to consider

uniq
Given a list L, create a new list L' that contains only the unique elements of L, in the order they appear in L.

sublists
Given a list L, generate a list L' of all sublists that can be made from the elements of L (elements must appear in same order).

reachable?
Given a graph G, starting point a and ending point b, determine whether b is reachable from a in G.
> (uniq '(california))
>'(california)

> (uniq '(mudd))
>'(mudd)

> (uniq '(mississippi))
>'(mississippi)
> (sublists '())
'(())

> (sublists '(1))
'(('1) ())

> (sublists '(1 2))
'(('1 2) (1) (2) ())
> (reachable? 'C 'A)  #t

> (reachable? 'A 'C)  #f
How hard are these problems?

<table>
<thead>
<tr>
<th>cost metric</th>
<th>worst-case input</th>
<th>worst-case cost</th>
</tr>
</thead>
<tbody>
<tr>
<td>uniq</td>
<td>number of comparisons made</td>
<td></td>
</tr>
<tr>
<td>sublists</td>
<td>number of sublists created</td>
<td></td>
</tr>
<tr>
<td>reachable?</td>
<td>number of edges traversed</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Cost Metric</th>
<th>Worst-Case Input</th>
<th>Worst-Case Cost</th>
</tr>
</thead>
<tbody>
<tr>
<td>uniq</td>
<td>number of comparisons made</td>
<td>all are unique</td>
</tr>
<tr>
<td>sublists</td>
<td>number of sublists created</td>
<td>everything is the worst (and the best!)</td>
</tr>
<tr>
<td>reachable?</td>
<td>number of edges traversed</td>
<td>b is not reachable from a</td>
</tr>
</tbody>
</table>
“Use it or lose it”
(a recursive search technique)
Review: recursion

A problem-solving strategy — fill in the **pieces**

**Base case(s)**

**Recursive case(s)**

1. **It:** A piece of the problem

2. **Lose-it solution:** How could we solve a smaller version *without* it?

3. How would we **combine** it and lose-it to solve the full problem?
> (sum '())
0

> (sum '(1 2 3))
6

> (sum '(7 7 7 7 7 7 7))
42
Review: recursion

a problem-solving strategy — fill in the pieces

Base case(s)

empty list → 0

Recursive case(s)

1. It: a piece of the problem
   the first element

2. Lose-it solution: How could we solve a smaller version without it?
   sum of the rest of the elements in the list

3. How would we combine it and lose-it to solve the full problem?
   add them
use it or lose it
a recursive strategy — fill in the pieces

Base case(s)

Recursive case(s)

1. **It**: a piece of the problem

2. **Lose-it solution**: How could we solve a smaller version *without* it?

3. **Use-it solution**: How could we solve a smaller version *with* it?

4. How would we **combine** the solutions to solve the full problem?
> (uniq 'california))
 'california

> (uniq 'mudd)
 'mudd

> (uniq 'mississippi)
 'mississippi
unique: use it or lose it

a recursive strategy — fill in the pieces

Base case(s)
empty list → empty list

Recursive case(s)
1. **It**: a piece of the problem
   the first element
2. **Smaller version**: What does the input look like without it?
   the rest of the list
3. **Lose-it solution**: How could we solve a smaller version *without* it?
   unique-ify the rest of the list
4. **Use-it solution**: How could we solve a smaller version *with* it?
   pre-pend “it” to the “lose-it” solution
5. How would we **combine** the solutions to solve the full problem?
   if “it” is in “lose-it”, then “lose-it”; else “use-it”

> (uniq '(c a l i f o r n i a))
'(c l f o r n i a)

> (uniq '(m u d d))
'(m u d)

> (uniq '(m i s s i p p i))
'(m s p i)
unique: use it or lose it

a recursive strategy

\[
\text{(define (uniq L)}
\]
unique: use it or lose it

a recursive strategy

(define (uniq L)
  (if (empty? L)
      ()
      (let* ([it  (first L)]
              [lose-it (uniq (rest L))]
              [use-it  (cons it lose-it)])
        (if (member it lose-it)
            lose-it
            use-it))))
> (sublists '())
  '(())

> (sublists '(1))
  '((1) ())

> (sublists '(1 2))
  '(((1 2) (1) (2) ())())
sublists: use it or lose it

a recursive strategy — fill in the pieces

Base case(s)

empty list → list of empty list

Recursive case(s)

1. **It:** a piece of the problem
   - the first element

2. **Smaller version:** What does the input look like without it?
   - the rest of the list

3. **Lose-it solution:** How could we solve a smaller version *without* it?
   - find sublists the rest of the list

4. **Use-it solution:** How could we solve a smaller version *with* it?
   - pre-pend "it" to each element of the "lose-it" solution

5. How would we **combine** the solutions to solve the full problem?
   - append the lose-it solution to the use-it solution

> (sublists '())
'(())

> (sublists '(1))
'((1) ())

> (sublists '(1 2))
'((1 2) (1) (2) ())
Solution technique: “use it or lose it”
a recursive search strategy

(define (sublists L)
  (if (empty? L)
    '(empty)
    (let* ([it (first L)]
            [lose-it (sublists (rest L))]
            [use-it (map (lambda (l) (cons it l)) lose-it)])
      (append use-it lose-it))))
reachable?: use it or lose it
a recursive strategy — fill in the pieces

Base case(s)
  a node is always reachable from itself
  no nodes are reachable in the empty graph

Recursive case(s)
1. **It:** a piece of the problem
   
an edge, \( c \rightarrow d \)
2. **Smaller version:** What does the input look like without it?
   
   the graph without that edge, called “sub-graph”
3. **Lose it:** How could we solve a smaller version *without* it?
   
is a reachable from \( b \) in sub-graph?
4. **Use it:** How could we solve a smaller version *with* it?
   
is \( c \) reachable from \( a \) AND is \( b \) reachable from \( d \) in sub-graph?
5. How would we **combine** the solutions to solve the full problem?
   
   use-it OR lose-it