Welcome to CS166!
Why study data structures?
Why Study Data Structures?

- **Expand your library of problem-solving tools.**
  - We’ll cover a wide range of tools for a bunch of interesting problems. These come in handy, both IRL an in Theoryland.

- **Learn new problem-solving techniques.**
  - We’ll see some truly beautiful problem-solving strategies that work beyond just a single example.

- **Challenge your intuition for the limits of efficiency.**
  - You'd be amazed how many times we'll take a problem you're sure you know how to solve and then see how to solve it faster.

- **See the beauty of theoretical computer science.**
  - We'll cover some amazingly clever theoretical techniques in the course of this class. You'll love them.
Where is CS166 situated in Stanford’s CS sequence?
**CS103**

\[ a_0 = 1 \quad a_{n+1} = 2a_n + n \]

**Theorem:** \( a_n = 2^{n+1} - n - 1 \).

**Proof:** By induction. As a base case, when \( n = 0 \), we have

\[
2^{n+1} - n - 1 = 2^1 - 0 - 1 = 1 = a_0.
\]

For the inductive step, assume that \( a_k = 2^{k+1} - k - 1 \). Then

\[
a_{k+1} = 2a_k + k = 2^{k+2} - 2k - 2 + k = 2^{(k+1)+1} - (k+1) - 1,
\]

as required. ■

**CS109**

\[
E \left[ \sum_{i=1}^{n} X_i \right] = \sum_{i=1}^{n} E[X_i]
\]

\[
Pr[X \geq c] \leq \frac{E[X]}{c}
\]

**CS161**

\[
T(n) = aT(n / b) + O(n^d)
\]

\[
n^2 \log n^2 = O(n^3)
\]

\[
n^2 \log n^2 = \Omega(n^2)
\]

\[
n^2 \log n^2 = \Theta(n^2 \log n)
\]
Who are we?
Course Staff

Keith Schwarz (htiek@cs.stanford.edu)

Francisco Pernice
Jose Calinawan Francisco

*Ping us over EdStem with questions!*
The Course Website

https://cs166.stanford.edu
Course Requirements

• We plan on having four *problem sets*.
  • Problem sets may be completed individually or in a pair.
  • They’re a mix of written problems and C++ coding exercises.
  • You’ll submit one copy of the problem set regardless of how many people worked on it.
  • Need to find a partner? Use EdStem, stop by office hours, or send us an email.

• We plan of having five *individual assessments*.
  • Similar to problem sets, except that they must be completed individually.
  • Course staff can answer clarifying questions, but otherwise it’s up to you to work out how to solve them.

• We plan to have a final *research project*.
  • We’ll hammer out details in the next couple of weeks. Expect to work in a group, do a deep dive into a topic, and get lots of support from us.
Individual Assessment 0

• Individual Assessment 0 goes out today. It’s due next Tuesday at 3:15PM Pacific time.

• This is mostly designed as a refresher of topics from the prerequisite courses CS103, CS107, CS109, and CS161.

• If you’re mostly comfortable with these problems and are just “working through some rust,” then you’re probably in the right place!
Let’s Get Started!
Range Minimum Queries
The RMQ Problem

- The **Range Minimum Query problem** (RMQ for short) is the following:

  Given an array A and two indices $i \leq j$, what is the smallest element out of $A[i], A[i + 1], \ldots, A[j - 1], A[j]$?
The RMQ Problem

• The **Range Minimum Query problem** (**RMQ** for short) is the following:

  Given an array $A$ and two indices $i \leq j$, what is the smallest element out of $A[i], A[i + 1], \ldots, A[j - 1], A[j]$?

• Notation: We'll denote a range minimum query in array $A$ between indices $i$ and $j$ as $\text{RMQ}_A(i, j)$.

• For simplicity, let's assume 0-indexing.
A Trivial Solution

- There's a simple $O(n)$-time algorithm for evaluating $\text{RMQ}_A(i, j)$: just iterate across the elements between $i$ and $j$, inclusive, and take the minimum!

- So... why is this problem at all algorithmically interesting?

- Suppose that the array $A$ is fixed in advance and you're told that we're going to make multiple queries on it.

- Can we do better than the naïve algorithm?
An Observation

- In an array of length $n$, there are only $\Theta(n^2)$ distinct possible queries.
- Why?

1 subarray of length 5
2 subarrays of length 4
3 subarrays of length 3
4 subarrays of length 2
5 subarrays of length 1
A Different Approach

- There are only $\Theta(n^2)$ possible RMQs in an array of length $n$.
- If we precompute all of them, we can answer RMQ in time $O(1)$ per query.
Building the Table

• One simple approach: for each entry in the table, iterate over the range in question and find the minimum value.

• How efficient is this?
  • Number of entries: $\Theta(n^2)$.
  • Time to evaluate each entry: $O(n)$.
  • Time required: $O(n^3)$.

• The runtime is $O(n^3)$ using this approach. Is it also $\Theta(n^3)$?
Each entry in yellow requires at least $n / 2 = \Theta(n)$ work to evaluate.

There are roughly $n^2 / 8 = \Theta(n^2)$ entries here.

Total work required: $\Theta(n^3)$
A Different Approach

• Naïvely precomputing the table is inefficient.
• Can we do better?
• **Claim:** We can precompute all subarrays in time $\Theta(n^2)$ using dynamic programming.
A Different Approach

- Naïvely precomputing the table is inefficient.
- Can we do better?
- **Claim:** We can precompute all subarrays in time $\Theta(n^2)$ using dynamic programming.
A Different Approach

- Naïvely precomputing the table is inefficient.
- Can we do better?
- **Claim:** We can precompute all subarrays in time $\Theta(n^2)$ using dynamic programming.
A Different Approach

- Naïvely precomputing the table is inefficient.
- Can we do better?
- **Claim:** We can precompute all subarrays in time $\Theta(n^2)$ using dynamic programming.
A Different Approach

- Naïvely precomputing the table is inefficient.
- Can we do better?
- **Claim:** We can precompute all subarrays in time $\Theta(n^2)$ using dynamic programming.
A Different Approach

• Naïvely precomputing the table is inefficient.
• Can we do better?
• **Claim:** We can precompute all subarrays in time $\Theta(n^2)$ using dynamic programming.
A Different Approach

- Naïvely precomputing the table is inefficient.
- Can we do better?
- **Claim:** We can precompute all subarrays in time $\Theta(n^2)$ using dynamic programming.
Some Notation

- We'll say that an RMQ data structure has time complexity \( \langle p(n), q(n) \rangle \) if
  - preprocessing takes time at most \( p(n) \) and
  - queries take time at most \( q(n) \).
- We now have two RMQ data structures:
  - \( \langle O(1), O(n) \rangle \) with no preprocessing.
  - \( \langle O(n^2), O(1) \rangle \) with full preprocessing.
- These are two extremes on a curve of tradeoffs: no preprocessing versus full preprocessing.
- **Question:** Is there a “golden mean” between these extremes?
Another Approach: **Block Decomposition**
A Block-Based Approach

- Split the input into $O(n / b)$ blocks of some “block size” $b$.
  - Here, $b = 4$.
- Compute the minimum value in each block.

<table>
<thead>
<tr>
<th></th>
<th>26</th>
<th></th>
<th>53</th>
<th></th>
<th>23</th>
<th></th>
<th>27</th>
<th></th>
<th>2</th>
</tr>
</thead>
<tbody>
<tr>
<td>31</td>
<td>41</td>
<td>59</td>
<td>26</td>
<td>53</td>
<td>58</td>
<td>97</td>
<td>93</td>
<td>23</td>
<td>84</td>
</tr>
<tr>
<td>62</td>
<td>43</td>
<td>33</td>
<td>83</td>
<td>27</td>
<td>95</td>
<td>2</td>
<td>88</td>
<td>41</td>
<td>97</td>
</tr>
</tbody>
</table>
Analyzing the Approach

- Let's analyze this approach in terms of $n$ and $b$.
- Preprocessing time:
  - $O(b)$ work on $O(n / b)$ blocks to find minima.
  - Total work: $O(n)$.
- Time to evaluate $\text{RMQ}_A(i, j)$:
  - $O(1)$ work to find block indices (divide by block size).
  - $O(b)$ work to scan inside $i$ and $j$'s blocks.
  - $O(n / b)$ work looking at block minima between $i$ and $j$.
  - Total work: $O(b + n / b)$. 
Intuiting $O(b + \frac{n}{b})$

- As $b$ increases:
  - The $b$ term rises (more elements to scan within each block).
  - The $\frac{n}{b}$ term drops (fewer blocks to look at).
- As $b$ decreases:
  - The $b$ term drops (fewer elements to scan within a block).
  - The $\frac{n}{b}$ term rises (more blocks to look at).
- Is there an optimal choice of $b$ given these constraints?
Optimizing $b$

- What choice of $b$ minimizes $b + n / b$?

Formulate a hypothesis, but *don’t post anything in chat just yet.*
Optimizing $b$

• What choice of $b$ minimizes $b + n / b$?

Now, *private chat me your best guess.*

Not sure? Just answer “??”
Optimizing $b$

- What choice of $b$ minimizes $b + n / b$?
- Start by taking the derivative:
  \[
  \frac{d}{db}(b+n/b) = 1 - \frac{n}{b^2}
  \]
- Setting the derivative to zero:
  \[
  1 - \frac{n}{b^2} = 0 \\
  1 = \frac{n}{b^2} \\
  b^2 = n \\
  b = \sqrt{n}
  \]
- Asymptotically optimal runtime is when $b = n^{1/2}$.
- In that case, the runtime is
  \[
  O(b + n / b) = O(n^{1/2} + n / n^{1/2}) = O(n^{1/2} + n^{1/2}) = O(n^{1/2})
  \]
Summary of Approaches

• Three solutions so far:
  • Full preprocessing: $\langle O(n^2), \ O(1) \rangle$.
  • Block partition: $\langle O(n), \ O(n^{1/2}) \rangle$.
  • No preprocessing: $\langle O(1), \ O(n) \rangle$.

• Modest preprocessing yields modest performance increases.

• **Question:** Can we do better?
A Second Approach: *Sparse Tables*
An Intuition

- The \( \langle O(n^2), O(1) \rangle \) solution gives fast queries because every range we might look up has already been precomputed.

- This solution is slow overall because we have to compute the minimum of every possible range.

- **Question:** Can we still get constant-time queries without preprocessing all possible ranges?
An Observation

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
</tr>
</thead>
<tbody>
<tr>
<td>31</td>
<td>31</td>
<td>31</td>
<td>26</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>41</td>
<td>41</td>
<td>26</td>
<td>26</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
| 2 |   | 59 | 26 | 26 | 26 |   |   | ★
| 3 |   |   | 26 | 26 | 26 | 26 |   |   |
| 4 |   |   |   | 53 | 53 | 53 | 53 |   |
| 5 |   |   |   |   | 58 | 58 | 58 |   |
| 6 |   |   |   |   |   | 97 | 93 |   |
| 7 |   |   |   |   |   |   | 93 |   |

- Blue boxes: Numbers divisible by 3
- Yellow box: Number 97
- Star: Number 41

Red lines indicate the following
- Horizontal: Numbers divisible by 3
- Vertical: The sequence of numbers from 0 to 7
The Intuition

- It's still possible to answer any query in time $O(1)$ without precomputing RMQ over all ranges.
- If we precompute the answers over too many ranges, the preprocessing time will be too large.
- If we precompute the answers over too few ranges, the query time won't be $O(1)$.
- **Goal:** Precompute RMQ over a set of ranges such that
  - There are $o(n^2)$ total ranges, but
  - there are enough ranges to support $O(1)$ query times.
The Approach

- For each index $i$, compute RMQ for ranges starting at $i$ of size 1, 2, 4, 8, 16, ..., $2^k$ as long as they fit in the array.
  - Gives both large and small ranges starting at any point in the array.
  - Only $O(\log n)$ ranges computed for each array element.
  - Total number of ranges: $O(n \log n)$.
- **Claim:** Any range in the array can be formed as the union of two of these ranges.
Creating Ranges
Creating Ranges

- 7
- 4
- 4
Doing a Query

- To answer $\text{RMQ}_A(i, j)$:
  - Find the largest $k$ such that $2^k \leq j - i + 1$.
    - With the right preprocessing, this can be done in time $O(1)$; you'll figure out how in an upcoming assignment.
  - The range $[i, j]$ can be formed as the overlap of the ranges $[i, i + 2^k - 1]$ and $[j - 2^k + 1, j]$.
  - Each range can be looked up in time $O(1)$.
  - Total time: $O(1)$. 
Precomputing the Ranges

- There are $O(n \log n)$ ranges to precompute.
- Using dynamic programming, we can compute all of them in time $O(n \log n)$. 
Precomputing the Ranges

- There are $O(n \log n)$ ranges to precompute.
- Using dynamic programming, we can compute all of them in time $O(n \log n)$.

```
<table>
<thead>
<tr>
<th>31</th>
<th>41</th>
<th>59</th>
<th>26</th>
<th>53</th>
<th>58</th>
<th>97</th>
<th>93</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
</tr>
</tbody>
</table>
```
Precomputing the Ranges

- There are $O(n \log n)$ ranges to precompute.
- Using dynamic programming, we can compute all of them in time $O(n \log n)$.
Precomputing the Ranges

- There are $O(n \log n)$ ranges to precompute.
- Using dynamic programming, we can compute all of them in time $O(n \log n)$.
Precomputing the Ranges

- There are $O(n \log n)$ ranges to precompute.
- Using dynamic programming, we can compute all of them in time $O(n \log n)$.
Sparse Tables

- This data structure is called a *sparse table*.
- It gives an $\langle O(n \log n), O(1) \rangle$ solution to RMQ.
- This is asymptotically better than precomputing all possible ranges!
The Story So Far

- We now have the following solutions for RMQ:
  - Precompute all: \( \langle O(n^2), \ O(1) \rangle \).
  - Sparse table: \( \langle O(n \log n), \ O(1) \rangle \).
  - Blocking: \( \langle O(n), \ O(n^{1/2}) \rangle \).
  - Precompute none: \( \langle O(1), \ O(n) \rangle \).

- **Can we do better?**
A Third Approach: *Hybrid Strategies*
Blocking Revisited
This is just RMQ on the block minima!
This is just RMQ inside the blocks!
The Framework

- Split the input into blocks of size $b$.
- Form an array of the block minima.
- Construct a “summary” RMQ structure over the block minima.
- Construct “block” RMQ structures for each block.
- Aggregate the results together.
Analyzing Efficiency

- Suppose we use a \((p_1(n), q_1(n))\)-time RMQ for the summary RMQ and a \((p_2(n), q_2(n))\)-time RMQ for each block, with block size \(b\).
- What is the preprocessing time for this hybrid structure?
  - \(O(n)\) time to compute the minima of each block.
  - \(O(p_1(n / b))\) time to construct RMQ on the minima.
  - \(O((n / b) p_2(b))\) time to construct the block RMQs.
- Total construction time is \(O(n + p_1(n / b) + (n / b) p_2(b))\).

Block size: \(b\).
# Blocks: \(O(n / b)\).
Analyzing Efficiency

- Suppose we use a \( p_1(n), q_1(n) \)-time RMQ for the summary RMQ and a \( p_2(n), q_2(n) \)-time RMQ for each block, with block size \( b \).
- What is the query time for this hybrid structure?
  - \( O(q_1(n / b)) \) time to query the summary RMQ.
  - \( O(q_2(b)) \) time to query the block RMQs.
- Total query time: \( O(q_1(n / b) + q_2(b)) \).

Block size: \( b \).
# Blocks: \( O(n / b) \).
Analyzing Efficiency

- Suppose we use a \( p_1(n), q_1(n) \)-time RMQ for the summary RMQ and a \( p_2(n), q_2(n) \)-time RMQ for each block, with block size \( b \).

- Hybrid preprocessing time:

  \[
  O(n + p_1(n/b) + (n/b)p_2(b))
  \]

- Hybrid query time:

  \[
  O(q_1(n/b) + q_2(b))
  \]
A Sanity Check

- The $\langle O(n), O(n^{1/2}) \rangle$ block-based structure from earlier uses this framework with the $\langle O(1), O(n) \rangle$ no-preprocessing RMQ structure and $b = n^{1/2}$.

Do no further preprocessing than just computing the block minima.

Don’t do anything fancy per block. Just do linear scans over each of them.
A Sanity Check

- The \( \langle O(n), O(n^{1/2}) \rangle \) block-based structure from earlier uses this framework with the \( \langle O(1), O(n) \rangle \) no-preprocessing RMQ structure and \( b = n^{1/2} \).

- According to our formulas, the preprocessing time should be

\[
O(n + p_1(n / b) + (n / b) p_2(b)) = O(n + 1 + n / b) = O(n)
\]

- The query time should be

\[
O(q_1(n / b) + q_2(b)) = O(n / b + b) = O(n^{1/2})
\]

- Looks good so far!
An Observation

• We can use any data structures we’d like for the summary and block RMQs.

• Suppose we use an $\langle O(n \log n), O(1) \rangle$ sparse table for the summary RMQ.

• If the block size is $b$, the time to construct a sparse table over the $(n / b)$ blocks is $O((n / b) \log (n / b))$.

• **Cute trick:** If $b = \Theta(\log n)$, the time to construct a sparse table over the minima is

\[
O((n / \log n) \log (n / \log n))
\]

\[
= O((n / \log n) \log n) \quad (O \text{ is an upper bound})
\]

\[
= O(n). \quad (\text{logs cancel out})
\]
One Possible Hybrid

- Set the block size to log \( n \).
- Use a sparse table for the summary RMQ.
- Use the “no preprocessing” structure for each block.

![Diagram showing block sizes and RMQ handling]

Summary RMQ \((\text{Sparse table})\)

Handled via linear scan

Table lookups
One Possible Hybrid

- Set the block size to $\log n$.
- Use a sparse table for the summary RMQ.
- Use the “no preprocessing” structure for each block.
- Preprocessing time:
  
  $$O(n + p_1(n / b) + (n / b) p_2(b))$$  
  
  $$= O(n + n + n / b)$$  
  
  $$= O(n)$$

- Query time:
  
  $$O(q_1(n / b) + q_2(b))$$  
  
  $$= O(1 + b)$$  
  
  $$= O(\log n)$$

- An \(O(n), O(\log n)\) solution!
Another Hybrid

- Let's suppose we use the \(O(n \log n), O(1)\) sparse table for both the summary and block RMQ structures with a block size of \(\log n\).
Another Hybrid

- Let's suppose we use the \( \langle O(n \log n), O(1) \rangle \) sparse table for both the summary and block RMQ structures with a block size of \( \log n \).

- The preprocessing time is

\[
\begin{align*}
O(n + p_1(n/b) + (n/b) p_2(b)) \\
= O(n + n + (n/b) b \log b) \\
= O(n + n \log b) \\
= O(n \log \log n)
\end{align*}
\]

- The query time is

\[
\begin{align*}
O(q_1(n/b) + q_2(b)) \\
= O(1)
\end{align*}
\]

- We have an \( \langle O(n \log \log n), O(1) \rangle \) solution to RMQ!

For Reference

\[
\begin{align*}
p_1(n) &= O(n \log n) \\
q_1(n) &= O(1) \\
p_2(n) &= O(n \log n) \\
q_2(n) &= O(1) \\
b &= \log n
\end{align*}
\]
One Last Hybrid

- Suppose we use a sparse table for the summary RMQ and the \(O(n), O(\log n)\) solution for the block RMQs. Let's choose \(b = \log n\).

<p>| | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>31</td>
<td>26</td>
<td>23</td>
<td>62</td>
<td>27</td>
</tr>
</tbody>
</table>

Summary RMQ *(Sparse table)*

Table lookups

It's complicated.
One Last Hybrid

- Suppose we use a sparse table for the summary RMQ and the \( \langle O(n), O(\log n) \rangle \) solution for the block RMQs. Let's choose \( b = \log n \).

- The preprocessing time is

\[
\begin{align*}
O(n + p_1(n / b) + (n / b) p_2(b)) &= O(n + n + (n / b) b) \\
&= O(n)
\end{align*}
\]

- The query time is

\[
\begin{align*}
O(q_1(n / b) + q_2(b)) &= O(1 + \log b) \\
&= O(\log \log n)
\end{align*}
\]

- We have an \( \langle O(n), O(\log \log n) \rangle \) solution to RMQ!

For Reference

- \( p_1(n) = O(n \log n) \)
- \( q_1(n) = O(1) \)
- \( p_2(n) = O(n) \)
- \( q_2(n) = O(\log n) \)
- \( b = \log n \)
Where We Stand

- We've seen a bunch of RMQ structures today:
  - No preprocessing: $O(1), O(n)$
  - Full preprocessing: $O(n^2), O(1)$
  - Block partition: $O(n), O(n^{1/2})$
  - Sparse table: $O(n \log n), O(1)$
  - Hybrid 1: $O(n), O(\log n)$
  - Hybrid 2: $O(n \log \log n), O(1)$
  - Hybrid 3: $O(n), O(\log \log n)$
Where We Stand

We've seen a bunch of RMQ structures today:

- **No preprocessing**: $\langle O(1), O(n) \rangle$
- **Full preprocessing**: $\langle O(n^2), O(1) \rangle$
- **Block partition**: $\langle O(n), O(n^{1/2}) \rangle$
- **Sparse table**: $\langle O(n \log n), O(1) \rangle$
- **Hybrid 1**: $\langle O(n), O(\log n) \rangle$
- **Hybrid 2**: $\langle O(n \log \log n), O(1) \rangle$
- **Hybrid 3**: $\langle O(n), O(\log \log n) \rangle$
Where We Stand

We've seen a bunch of RMQ structures today:

No preprocessing: $\langle O(1), O(n) \rangle$

Full preprocessing: $\langle O(n^2), O(1) \rangle$

- **Block partition**: $\langle O(n), O(n^{1/2}) \rangle$
  
  Sparse table: $\langle O(n \log n), O(1) \rangle$

- **Hybrid 1**: $\langle O(n), O(\log n) \rangle$

- **Hybrid 2**: $\langle O(n \log \log n), O(1) \rangle$

- **Hybrid 3**: $\langle O(n), O(\log \log n) \rangle$
Is there an $\langle O(n), O(1) \rangle$ solution to RMQ?

Yes!
Next Time

- **Cartesian Trees**
  - A data structure closely related to RMQ.

- **The Method of Four Russians**
  - A technique for shaving off log factors.

- **The Fischer-Heun Structure**
  - A clever, asymptotically optimal RMQ structure.