Announcements

● PA0 has been released...start early
  ○ PA1 will be released this week
● Office hours and recitations start this week
From Lecture 01...

Option 1
- Very fast Prepend, Get First
- Very slow Get Nth

Option 2
- Very fast Get Nth, Get First
- Very slow Prepend

Option 3
- Very fast Get Nth, Get First
- Occasionally slow Prepend
From Lecture 01...

Option 1
- Very fast Prepend, Get First
- Very slow Get Nth

Option 2
- Very fast Get Nth, Get First
- Very slow Prepend

Option 3
- Very fast Get Nth, Get First
- Occasionally slow Prepend

What is fast? slow?
Attempt #1: Wall-clock time?

- What is fast?
  - 10s? 100ms? 10ns?
  - ...it depends on the task
- **Algorithm vs Implementation**
  - Compare Grace Hopper's implementation to yours
- What machine are you running on?
  - Your old laptop? A lab machine? The newest, shiniest processor?
- What bottlenecks exist? CPU vs IO vs Memory vs Network...
Attempt #1: Wall-clock time?

- What is fast?
  - 10s? 100ms? 10ns?
  - ...it depends on the task
- Algorithm vs Implementation
  - Compare Grace Hopper’s implementation to yours
- What machine are you running on?
  - Your old laptop? A lab machine? The newest, shiniest processor?
- What bottlenecks exist? CPU vs IO vs Memory vs Network…

Wall-clock time is not terribly useful...
Let’s do a quick demo...
Comparing Random Access for Array vs List

**Array**

**List**

---

8
Comparing Random Access for Array vs List

Let’s ignore the specific numbers and clean things up a bit...
Comparing Random Access for Array vs List
Comparing Random Access for Array vs List

What differentiates these two algorithms is how they scale with input size (the shape of the function)
When is an algorithm “fast”?

- To give a useful solution, we should take “scale” into account
  - How does the runtime change as we change the size of the input (number of users, records, pixels, elements, etc)
- Don’t think in terms of wall-time, think in terms of “number of steps”
Scaling Examples

- “Five steps plus Ten steps per user”
- “Ten steps per network connection. Each node has connections to 1% of the other nodes in the system”
- “Seven steps for every possible pair of elements”
- “For each user, Ten steps, plus Three steps per post”
Scaling Examples

● “Five steps plus Ten steps per user”
  ○ \(5 + (10 \times |\text{Users}|)\)

● “Ten steps per network connection. Each node has connections to 1% of the other nodes in the system”

● “Seven steps for every possible pair of elements”

● “For each user, Ten steps, plus Three steps per post”
Scaling Examples

● “Five steps plus Ten steps per user”
  ○ $5 + (10 \times |\text{Users}|)$

● “Ten steps per network connection. Each node has connections to 1% of the other nodes in the system”
  ○ $10 \times (|\text{Nodes}| \times (0.01 \times |\text{Nodes}|))$

● “Seven steps for every possible pair of elements”

● “For each user, Ten steps, plus Three steps per post”
Scaling Examples

- “Five steps plus Ten steps per user”
  - $5 + (10 \times |\text{Users}|)$
- “Ten steps per network connection. Each node has connections to 1% of the other nodes in the system”
  - $10 \times (|\text{Nodes}| \times (0.01 \times |\text{Nodes}|))$
- “Seven steps for every possible pair of elements”
  - $7 \times 2^{|\text{Elements}|}$
- “For each user, Ten steps, plus Three steps per post”
Scaling Examples

- “Five steps plus Ten steps per user”
  - $5 + (10 \times |\text{Users}|)$

- “Ten steps per network connection. Each node has connections to 1% of the other nodes in the system”
  - $10 \times (|\text{Nodes}| \times (0.01 \times |\text{Nodes}|))$

- “Seven steps for every possible pair of elements”
  - $7 \times 2^{|\text{Elements}|}$

- “For each user, Ten steps, plus Three steps per post”
  - $|\text{Users}| \times (10 + 3 \times |\text{Posts}|)$
Would you consider an algorithm that takes $|\text{Users}|!$ number of steps?
Would you consider an algorithm that takes $|\text{Users}|!$ number of steps?
Would you consider an algorithm that takes $|\text{Users}|!$ number of steps?

NO!

maybe...
Runtime as a Function

Which is better? $3x|\text{Users}| + 5$ or $|\text{Users}|^2$
When is an algorithm “fast”? 

- To give a useful solution, we should take “scale” into account
  - How does the runtime change as we change the size of the input (number of users, records, pixels, elements, etc)
- Don’t think in terms of wall-time, think in terms of “number of steps”
When is an algorithm “fast”?

- To give a useful solution, we should take “scale” into account
  - How does the runtime change as we change the size of the input (number of users, records, pixels, elements, etc)
- Don’t think in terms of wall-time, think in terms of “number of steps”
- **Focus on “large” inputs**
  - Rank functions based on how they behave at large scales
Which is better? $3x|\text{Users}|+5$ or $|\text{Users}|^2$
Goal: Ignore implementation details

Seasoned Pro Implementation vs Error 23: Cat on Keyboard
Goal: Ignore execution environment

Intel i9 vs Motorola 68000
Goal: Judge the Algorithm Itself

- How fast is a step? Don’t care
  - Only count number of steps
- Can this be done in two steps instead of one?
  - “3 steps per user” vs “some number of steps per user”
  - Sometimes we don’t care...sometimes we do
When is an algorithm “fast”?

- To give a useful solution, we should take “scale” into account
  - How does the runtime change as we change the size of the input (number of users, records, pixels, elements, etc)
- Don’t think in terms of wall-time, think in terms of “number of steps”
- Focus on “large” inputs
  - Rank functions based on how they behave at large scales
When is an algorithm “fast”?

● To give a useful solution, we should take “scale” into account
  ○ How does the runtime change as we change the size of the input (number of users, records, pixels, elements, etc)
● Don’t think in terms of wall-time, think in terms of “number of steps”
● Focus on “large” inputs
  ○ Rank functions based on how they behave at large scales
● Decouple algorithm from infrastructure/implementation
  ○ Asymptotic notation...?
And now a brief interlude...
Logarithms (refresher)

Let \( a, b, c, n > 0 \)

**Exponent Rule:** \( \log(n^a) = a \log(n) \)

**Product Rule:** \( \log(an) = \log(a) + \log(n) \)

**Division Rule:** \( \log(n/a) = \log(n) - \log(a) \)

**Change of Base:** \( \log_b(n) = \log_c(n) / \log_b(n) \)

**Log/Exponent are Inverses:** \( b^{\log(n)} = \log_b(b^n) = n \)
Logarithms (refresher)

Let $a, b, c, n > 0$

**Exponent Rule:** $\log(n^a) = a \log(n)$

**Product Rule:** $\log(an) = \log(a) + \log(n)$

**Division Rule:** $\log(n/a) = \log(n) - \log(a)$

**Change of Base:** $\log_b(n) = \log_c(n) / \log_b(n)$

**Log/Exponent are Inverses:** $b^{\log(n)} = \log_b(b^n) = n$

In this class, always assume $\log$ base 2 unless specified otherwise
Now back to “fast”...
Attempt #2: Growth Functions

Not a function in code...but a mathematical function:

\[ f(n) \]

\( n \): The “size” of the input

ie: number of users, rows, pixels, etc

\( f(n) \): The number of “steps” taken for input of size \( n \)

ie: 20 steps per user, where \( n = |\text{Users}| \), is \( 20 \times n \)
Some Basic Assumptions:

Problem sizes are non-negative integers

\[ n \in \mathbb{Z}^+ \cup \{0\} = \{0, 1, 2, 3, \ldots\} \]

We can’t reverse time...(obviously)

\[ f(n) > 0 \]

Smaller problems aren’t harder than bigger problems

\[ \forall n_1 < n_2, f(n_1) \leq f(n_2) \]
Some Basic Assumptions:

Problem sizes are non-negative integers

\[ n \in \mathbb{Z}^+ \cup \{0\} = \{0, 1, 2, 3, \ldots\} \]

We can’t reverse time...(obviously)

\[ f(n) > 0 \]

Smaller problems aren’t harder than bigger problems

\[ \forall n_1 < n_2, \ f(n_1) \leq f(n_2) \]
First Problem...

We are still implementation dependent

\[ f_1(n) = 20n \]
\[ f_2(n) = 19n \]
First Problem...

We are still implementation dependent

\[ f_1(n) = 20n \]
\[ f_2(n) = 19n \]

Does 1 extra step per element really matter...?
First Problem...

We are still implementation dependent.

\[ f_1(n) = 20n \]
\[ f_2(n) = 19n \]
\[ f_3(n) = \frac{n^2}{2} \]

\( f_1 \) and \( f_2 \) are much more “similar” to each other than they are to \( f_3 \).
How Do We Capture Behavior at Scale?

Consider the following two functions:

\[
\frac{1}{100}n^3 + 10n + 1000000 \log(n)
\]
How Do We Capture Behavior at Scale?
How Do We Capture Behavior at Scale?

After this point, these functions behave the same (they stay about 100x apart)
How Do We Capture Behavior at Scale?

\[
\lim_{n \to \infty} \frac{\frac{1}{100} n^3 + 10n + 1000000 \log(n)}{n^3}
\]
How Do We Capture Behavior at Scale?

\[
\lim_{n \to \infty} \frac{\frac{1}{100} n^3 + 10n + 1000000 \log(n)}{n^3} = \lim_{n \to \infty} \frac{\frac{1}{100} n^3}{n^3} + \frac{10n}{n^3} + \frac{1000000 \log(n)}{n^3}
\]
How Do We Capture Behavior at Scale?

\[
\lim_\limits_{n \to \infty} \frac{\frac{1}{100} n^3 + 10n + 1000000 \log(n)}{n^3} = \lim_\limits_{n \to \infty} \frac{\frac{1}{100} n^3}{n^3} + \frac{10n}{n^3} + \frac{1000000 \log(n)}{n^3} = \lim_\limits_{n \to \infty} \frac{1}{100} + \lim_\limits_{n \to \infty} \frac{10}{n^2} + \lim_\limits_{n \to \infty} \frac{1000000 \log(n)}{n^3}
\]

These terms go to 0
How Do We Capture Behavior at Scale?

\[
\lim_{n \to \infty} \frac{\frac{1}{100} n^3 + 10n + 1000000 \log(n)}{n^3} = \lim_{n \to \infty} \frac{\frac{1}{100} n^3}{n^3} + \frac{10n}{n^3} + \frac{1000000 \log(n)}{n^3} = \lim_{n \to \infty} \frac{1}{100} + \lim_{n \to \infty} \frac{10}{n^2} + \lim_{n \to \infty} \frac{1000000 \log(n)}{n^3} = \frac{1}{100}
\]
Attempt #3: Asymptotic Analysis

Consider two functions, $f(n)$ and $g(n)$

$$\lim_{n \to \infty} \frac{f(n)}{g(n)} = \infty$$

In this particular case, $f$ grows w.r.t. $n$ faster than $g$

So...if $f(n)$ and $g(n)$ are the number of steps two different algorithms take on a problem of size $n$, which is better?
Attempt #3: Asymptotic Analysis

**Case 1:** \[ \lim_{n \to \infty} \frac{f(n)}{g(n)} = \infty \] (f grows faster; g is better)

**Case 2:** \[ \lim_{n \to \infty} \frac{f(n)}{g(n)} = 0 \] (g grows faster; f is better)

**Case 3:** \[ \lim_{n \to \infty} \frac{f(n)}{g(n)} = \text{some constant} \] (f and g “behave” the same)
Goal of “Asymptotic Analysis”

We want to organize runtimes (growth functions) into different Complexity Classes.

Within the same complexity class, runtimes “behave the same”
Goal of “Asymptotic Analysis”

“Strategic Optimization” focuses on improving the complexity class of your code!
Back to Our Previous Example...

\[
\frac{1}{100}n^3 + 10n + 1000000 \log(n)
\]

The $10n$ and $1000000 \log(n)$ “don’t matter”

The $1/100$ “does not matter”
Back to Our Previous Example...

\[ \frac{1}{100} n^3 + 10n + 1000000 \log(n) \]

The $10n$ and $1000000 \log(n)$ “don’t matter”

The $1/100$ “does not matter”

$n^3$ is the dominant term, and that determines the “behavior”
## Why Focus on Dominating Terms?

<table>
<thead>
<tr>
<th>( f(n) )</th>
<th>10</th>
<th>20</th>
<th>50</th>
<th>100</th>
<th>1000</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \log(\log(n)) )</td>
<td>0.43 ns</td>
<td>0.52 ns</td>
<td>0.62 ns</td>
<td>0.68 ns</td>
<td>0.82 ns</td>
</tr>
<tr>
<td>( \log(n) )</td>
<td>0.83 ns</td>
<td>1.01 ns</td>
<td>1.41 ns</td>
<td>1.66 ns</td>
<td>2.49 ns</td>
</tr>
<tr>
<td>( n )</td>
<td>2.5 ns</td>
<td>5 ns</td>
<td>12.5 ns</td>
<td>25 ns</td>
<td>0.25 µs</td>
</tr>
<tr>
<td>( n \log(n) )</td>
<td>8.3 ns</td>
<td>22 ns</td>
<td>71 ns</td>
<td>0.17 µs</td>
<td>2.49 µs</td>
</tr>
<tr>
<td>( n^2 )</td>
<td>25 ns</td>
<td>0.1 µs</td>
<td>0.63 µs</td>
<td>2.5 µs</td>
<td>0.25 ms</td>
</tr>
<tr>
<td>( n^5 )</td>
<td>25 µs</td>
<td>0.8 ms</td>
<td>78 ms</td>
<td>2.5 s</td>
<td>2.9 days</td>
</tr>
<tr>
<td>( 2^n )</td>
<td>0.25 µs</td>
<td>0.26 ms</td>
<td>3.26 days</td>
<td>10^{13} years</td>
<td>10^{284} years</td>
</tr>
<tr>
<td>( n! )</td>
<td>0.91 ms</td>
<td>19 years</td>
<td>10^{47} years</td>
<td>10^{141} years</td>
<td>10^{358} years</td>
</tr>
</tbody>
</table>
Why Focus on Dominating Terms?

\[ 2^n \gg n^c \gg n \gg \log(n) \gg c \]