Week 2.1, Monday, August 26

Homework 1 available on course web page
(Due: September 3 at 11:59PM on Gradescope)
Office Hours (Piazza)

- **Monday**
  - 9-10:30 AM (Hai Nguyen --- HAAS G050)
  - 2:30-3:30 PM (Prof. Blocki --- LWSN 1165)
  - 3:30-4:30 PM (Utkarsh Jain --- HAAS G050)

- **Tuesday**
  - 8-9:30AM (Noah Franks --- HAAS G050)
  - 11:45AM-1:15PM (Himanshi Mehta --- HAAS G050)

- **Wednesday**
  - 9:30 -11AM (Mike Cinkoske --- HAAS G050)
  - 11:30AM-1PM (Kevin Xia --- HAAS G050)
  - 2:30-3:30PM (Prof. Blocki --- LWSN 1165)
  - 4-5:30PM (Ahammed Ullah --- HAAS G050)
Office Hours (Piazza)

- **Thursday**
  - 8-9:30AM (Noah Franks --- HAAS G050)
  - 11:45-1:15AM (Himanshi Mehta --- HAAS G050)

- **Friday**
  - 10-11:30AM (Abhishek Sharma --- LWSN 3rd floor lobby*)
  - 2:30-3:30PM (Hiten Rathod --- HAAS G050)
  - 5-6:30PM (Tunaz Islam --- HAAS G050)

* If available; otherwise HAAS G050
What do we count?

- Time and space
  - time in terms of number of basic operations on basic data types
- Ignore machine dependent factors, but remain realistic
- Random Access Model (RAM)
  - no concurrency
  - count instructions (arithmetic operation, comparison, data movement)
  - each instruction takes constant time
  - realistic assumption on the size of the numbers (to represent $n$, it takes $\log n$ bits)
O(g(n)) = \{ f(n) | \text{ there exist positive constants } c \text{ and } n_0 \text{ such that } 0 \leq f(n) \leq c g(n) \text{ or all } n \geq n_0 \} \\

We write \( f(n) = O(g(n)) \) if there exist constants \( c > 0, n_0 > 0 \) such that \( 0 \leq f(n) \leq cg(n) \) for all \( n \geq n_0 \).

4n + 23\log n -28 = O(n)
- Drops low-order terms
- Ignores leading constants
- May not hold for small values of \( n \)
\( f(n) = O(g(n)) \) if there exist constants \( c > 0, \ n_0 > 0 \) such that \( 0 \leq f(n) \leq c g(n) \) for all \( n \geq n_0 \).

\[
f(n) = 3n^2 - 4n + 512 \\
\leq 3n^2 + 512 \\
\leq 4n^2 \text{ for } n \geq 23
\]

(set \( c=4, \ n_0 = 23 \))

- \( f(n) = O(n^2) \)
- \( f(n) = O(n^3) \) also holds
- \( f(n) = O(n) \) is false

CLRS text Figure 3.1
Which statements are true?

\[ 3n^3 + 90n^2 - 5n = O(n^3) \]
\[ 3n^3 + 90n^2 - 5n = O(2^n) \]
\[ 3n^3 + 90n^2 - 5n = O(n^2) \]
\[ 5 \log n = O(n) \]
\[ \sqrt{n} = O(\log n^8) \]
\[ n \log n = O(n) \]
\[ 4n = O(n \log n) \]
\[ n / \log n = O(\sqrt{n}) \]
Which statements are true?

\[3n^3 + 90n^2 - 5n = O(n^3)\] true
\[3n^3 + 90n^2 - 5n = O(2^n)\] true
\[3n^3 + 90n^2 - 5n = O(n^2)\] false
\[5 \log n = O(n)\] true
\[\sqrt{n} = O(\log n^8)\] false
\[n \log n = O(n)\] false
\[4n = O(n \log n)\] true
\[n/\log n = O(\sqrt{n})\] false
Consider two running times: \( 4n\log n \) and \( 8n^{n^{1/8}} \)

Which relationships hold?

1. \( 4n\log n = O(8n^{n^{1/8}}) \)
2. \( 8n^{n^{1/8}} = O(4n\log n) \)
3. \( 4n\log n = \Theta(8n^{n^{1/8}}) \)
4. \( 8n^{n^{1/8}} = \Theta(4n\log n) \)

A. None
B. 1
C. 2
D. 1 and 3
E. 4
Suppose that \( f, g, \) and \( h \) are positive functions (i.e., \( f(n), g(n), h(n) \geq 1 \) for all \( n \geq 1 \)). Which of the following claims are necessarily true?

1. \( f(n) = O(g(n)) \implies f(n) = O(f(n) + g(n)) \)
2. \( f(n) = O(g(n)) \) and \( g(n) = O(h(n)) \implies f(n) = O(h(n)) \)
3. \( f(n) = O(f(n/2)) \)

A. All of the above
B. 1
C. 2
D. 1 and 3
E. 1 and 2
Asymptotic Bounds

$O(g(n)) = \{ f(n) | \text{ there exist positive constants } c \text{ and } n_0 \text{ such that } 0 \leq f(n) \leq c \cdot g(n) \text{ for all } n \geq n_0 \}$

$O$ captures upper bounds

$\Theta(g(n)) = \{ f(n) | \text{ there exist positive constants } c_1, c_2, \text{ and } n_0 \text{ such that } 0 \leq c_1 \cdot g(n) \leq f(n) \leq c_2 \cdot g(n) \text{ for all } n \geq n_0 \}$

$\Theta$ captures upper and lower bounds
Examples

- $3n^3 + 90n^2 - 5n$ is $O(n^3)$ and $\Theta(n^3)$ is true
- $3n^3 + 90n^2 - 5n$ is $O(2^n)$ true, but $\Theta(2^n)$ false
- $5 \log n$ is $O(n)$ true, but $\Theta(n)$ false
- $4n = O(n \log n)$ is true, but $\Theta(n \log n)$ false
Asymptotic Bounds

\( \bigO(g(n)) = \{ f(n) \mid \text{there exist positive constants } c \text{ and } n_0 \text{ such that } 0 \leq f(n) \leq c \ g(n) \text{ for all } n \geq n_0 \} \)

\( \bigO \) captures upper bounds

\( \Theta(g(n)) = \{ f(n) \mid \text{there exist positive constants } c_1, c_2, \text{ and } n_0 \text{ such that } 0 \leq c_1 \ g(n) \leq f(n) \leq c_2 \ g(n) \text{ for all } n \geq n_0 \} \)

\( \Theta \) captures upper and lower bounds

\( \Omega(g(n)) = \{ f(n) \mid \text{there exist positive constants } c \text{ and } n_0 \text{ such that } 0 \leq c g(n) \leq f(n) \text{ for all } n \geq n_0 \} \)

\( \Omega \) captures lower bounds

\( 4n \log n = \Omega(n) \)
We will generally assume that \( n \) is “nice”
- E.g., power of 2
- We are not implementing the algorithms and only need to consider crucial the boundary/special cases

When asked to design an efficient algorithm
- sometimes you will be given a target asymptotic bound
  - other times you need to find the “best” one

You can use known data structures
- State how they are implemented and give time bounds of operations
How many times is F called?

Assume $n$ is a power of 4 ($n=4^k$)

```
while $n > 1$ do
  for $i = 1$ to $n$ do
    $F(i, n)$
  $n = n/4$
```

$O(n \log n)$  \(\Theta(n \log n)\)

$O(n^2)$  \(\Theta(n^2)\)

$O(n)$  \(\Theta(n)\)

$O(\log n)$  \(\Theta(\log n)\)
How many times is F called?

Assume \( n \) is a power of 4 \( (n=4^k) \)

\[
\text{while } n > 1 \text{ do}
\]
\[
\text{for } i = 1 \text{ to } n \text{ do}
\]
\[
F(i,n)
\]
\[
n = n/4
\]

\( O(n \log n) \quad \Theta(n \log n) \)
\( O(n^2) \quad \Theta(n^2) \)
\( O(n) \quad \Theta(n) \)
\( O(\log n) \quad \Theta(\log n) \)
Assume \( n \) is a power of 4 \((n=4^k)\)

\[
\sum_{i=1}^{k} 4^k
\]

\(\mathcal{O}(n \log n)\) \(\Theta(n \log n)\)

\(\mathcal{O}(n^2)\) \(\Theta(n^2)\)

\(\mathcal{O}(n)\) \(\Theta(n)\)

\(\mathcal{O}(\log n)\) \(\Theta(\log n)\)
Fact: Suppose $x \neq 1$ then $\sum_{i=0}^{k} x^i = \frac{x^{k+1} - 1}{x - 1}$

Proof: $\sum_{i=0}^{k} x^i = \frac{1-x}{1-x} \sum_{i=0}^{k} x^i$

$$= \frac{1}{1-x} \left( \sum_{i=0}^{k} x^i - \sum_{i=0}^{k} x^{i+1} \right)$$

$$= \frac{1}{1-x} \left( \sum_{i=0}^{k} x^i - \sum_{i=1}^{k+1} x^i \right) = \frac{x^{k+1} - 1}{x - 1}$$

Example: $x = 4$ we have $\sum_{i=0}^{k} 4^i = \frac{4^{k+1} - 1}{3} = \frac{4n-1}{3}$
Assume \( n \) is a power of 4 (\( n=4^k \))

\[
\text{while } n > 1 \text{ do}
\]
\[
\quad \text{for } i = 1 \text{ to } n \text{ do}
\]
\[
\quad \quad \text{F}(i, n)
\]
\[
\quad n = n/4
\]

\( \Theta(n \log n) \quad \Theta(n \log n) \)

\( \Theta(n^2) \quad \Theta(n^2) \)

\( \Theta(n) \quad \Theta(n) \)

\( \Theta(\log n) \quad \Theta(\log n) \)
**Common complexity classes**

\(O(1)\) – constant
\(O(\log n)\) – logarithmic (any base)
\(O(\log^k n)\) – poly log

\(O(n)\) – linear
\(O(n \log n)\) – quasi-linear
\(O(n^2)\) – quadratic
\(O(n^3)\) – cubic
\(O(n^k)\) – polynomial, \(k\) is a positive constant

\(O(2^n), O(c^n)\) – exponential, \(c\) is a constant > 1
\(O(n!)\) – factorial
\(O(n^n)\)
Table 2.1 The running times (rounded up) of different algorithms on inputs of increasing size, for a processor performing a million high-level instructions per second. In cases where the running time exceeds $10^{25}$ years, we simply record the algorithm as taking a very long time.

<table>
<thead>
<tr>
<th>$n$</th>
<th>$n$</th>
<th>$n \log_2 n$</th>
<th>$n^2$</th>
<th>$n^3$</th>
<th>$1.5^n$</th>
<th>$2^n$</th>
<th>$n!$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$n = 10$</td>
<td>$&lt; 1$ sec</td>
<td>$&lt; 1$ sec</td>
<td>$&lt; 1$ sec</td>
<td>$&lt; 1$ sec</td>
<td>$&lt; 1$ sec</td>
<td>$&lt; 1$ sec</td>
<td>4 sec</td>
</tr>
<tr>
<td>$n = 30$</td>
<td>$&lt; 1$ sec</td>
<td>$&lt; 1$ sec</td>
<td>$&lt; 1$ sec</td>
<td>$&lt; 1$ sec</td>
<td>$&lt; 1$ sec</td>
<td>18 min</td>
<td>$10^{25}$ years</td>
</tr>
<tr>
<td>$n = 50$</td>
<td>$&lt; 1$ sec</td>
<td>$&lt; 1$ sec</td>
<td>$&lt; 1$ sec</td>
<td>$&lt; 1$ sec</td>
<td>11 min</td>
<td>36 years</td>
<td>very long</td>
</tr>
<tr>
<td>$n = 100$</td>
<td>$&lt; 1$ sec</td>
<td>$&lt; 1$ sec</td>
<td>$&lt; 1$ sec</td>
<td>1 sec</td>
<td>12,892 years</td>
<td>$10^{17}$ years</td>
<td>very long</td>
</tr>
<tr>
<td>$n = 1,000$</td>
<td>$&lt; 1$ sec</td>
<td>$&lt; 1$ sec</td>
<td>1 sec</td>
<td>18 min</td>
<td>very long</td>
<td>very long</td>
<td>very long</td>
</tr>
<tr>
<td>$n = 10,000$</td>
<td>$&lt; 1$ sec</td>
<td>$&lt; 1$ sec</td>
<td>2 min</td>
<td>12 days</td>
<td>very long</td>
<td>very long</td>
<td>very long</td>
</tr>
<tr>
<td>$n = 100,000$</td>
<td>$&lt; 1$ sec</td>
<td>2 sec</td>
<td>3 hours</td>
<td>32 years</td>
<td>very long</td>
<td>very long</td>
<td>very long</td>
</tr>
<tr>
<td>$n = 1,000,000$</td>
<td>1 sec</td>
<td>20 sec</td>
<td>12 days</td>
<td>31,710 years</td>
<td>very long</td>
<td>very long</td>
<td>very long</td>
</tr>
</tbody>
</table>
The divide-and-conquer algorithm design paradigm

1. *Divide* the problem (instance) into subproblems.

2. *Conquer* the subproblems by solving them recursively.

3. *Combine* subproblem solutions.
Divide-and-Conquer

- Divide-and-conquer.
  - Break up problem into several parts.
  - Solve each part recursively.
  - Combine solutions to sub-problems into overall solution.

- Most common usage.
  - Break up problem of size $n$ into two equal parts of size $\frac{1}{2}n$.
  - Solve two parts recursively.
  - Combine two solutions into overall solution in linear time.

- Consequence.
  - Brute force: $n^2$.
  - Divide-and-conquer: $n \log n$.

- Divide et impera.
  Veni, vidi, vici.
  - Julius Caesar
Running Time (Recurrences):
- Let $T(n)$ be the time to solve problem of size $n$ (worst-case).
- Suppose we split input $X$ into 3 equal size parts $A$, $B$ and $C$ recursively solve smaller problems $A$, $B$ and $C$ and then merge the solutions.

$$T(n) \leq 3T\left(\frac{n}{3}\right) + \text{#Steps(Merge)}$$

Correctness?
- Induction!
- Prove that algorithm is correct on small inputs (e.g., $n \leq 2$)
- Prove that merge algorithm is correct (QED)
What you should learn?

- Solve Recurrences
- Identify recurrence associated with divide and conquer algorithm
- Prove that a divide and conquer algorithm is correct
- **Creative:** Design efficient divide and conquer algorithms
  - Build intuition about when the divide and conquer approach will work.
- **Mergesort.**
  - Divide array into two halves.
  - Recursively sort each half.
  - Merge two halves to make sorted whole.

![Diagram of Mergesort](image)

\[
T(n) \leq 2T\left(\frac{n}{2}\right) + O(n)
\]
Merging

- Merging. Combine two pre-sorted lists into a sorted whole.

- How to merge efficiently?
  - Linear number of comparisons.
  - Use temporary array.

- Challenge for the bored. In-place merge. [Kronrud, 1969]
  - using only a constant amount of extra storage
A Useful Recurrence Relation

- Def. $T(n) = \text{number of comparisons to mergesort an input of size } n$.

- Mergesort recurrence.

  $$T(n) \leq \begin{cases} 
  0 & \text{solve left half} \\
  T\left(\left\lceil \frac{n}{2} \right\rceil \right) + T\left(\left\lfloor \frac{n}{2} \right\rfloor \right) + n & \text{merging} \\
  \text{if } n = 1 \\
  \text{otherwise}
  \end{cases}$$

- Solution. $T(n) = \mathcal{O}(n \log_2 n)$.

- Assorted proofs. We describe several ways to prove this recurrence. Initially we assume $n$ is a power of 2 and replace $\leq$ with $=$.
Claim. If $T(n)$ satisfies this recurrence, then $T(n) = n \log_2 n$.

Pf. (by induction on $n$)

- Base case: $n = 1$.
- Inductive hypothesis: $T(n) = n \log_2 n$.
- Goal: show that $T(2n) = 2n \log_2 (2n)$.

\[
T(2n) = \begin{cases} 
0 & \text{if } n = 1 \\
2T(n/2) + n & \text{otherwise}
\end{cases}
\]

assumes $n$ is a power of 2

\[
T(2n) = 2T(n) + 2n \\
= 2n \log_2 n + 2n \\
= 2n(\log_2 (2n) - 1) + 2n \\
= 2n \log_2 (2n)
\]
Claim. If $T(n)$ satisfies the following recurrence, then $T(n) \leq n \lceil \lg n \rceil$.

\[
T(n) \leq \begin{cases} 
  0 & \text{if } n = 1 \\
  T(\lceil n/2 \rceil) + T(\lfloor n/2 \rfloor) + n & \text{otherwise}
\end{cases}
\]

Pf. (by induction on $n$)

- Base case: $n = 1$.
- Define $n_1 = \lfloor n / 2 \rfloor$, $n_2 = \lceil n / 2 \rceil$.
- Induction step: assume true for $1, 2, \ldots, n-1$. 
Problem: Compute $a^n$, $n>0$. Minimize number of multiplications.

Naive algorithm: $\Theta(n)$ multiplications

Divide-and-conquer algorithm:

$$a^n = \begin{cases} 
  a^{n/2} \cdot a^{n/2} & \text{if } n \text{ is even; } \\
  a^{(n-1)/2} \cdot a^{(n-1)/2} \cdot a & \text{if } n \text{ is odd. } 
\end{cases}$$

$$T(n) = T(n/2) + \Theta(1) \Rightarrow T(n) = \Theta(\log n)$$