Midterm Review

CS16: Introduction to Data Structures & Algorithms
Spring 2019
What is Running Time?

Worst-case running time

\[ T(n) : \text{Number of elementary operations on worst-case input as a function of input size } n \]
Q: how do we compare running times?
<table>
<thead>
<tr>
<th>$n$</th>
<th>log $n$</th>
<th>$n$</th>
<th>$n \log n$</th>
<th>$n^2$</th>
<th>$n^3$</th>
<th>$2^n$</th>
</tr>
</thead>
<tbody>
<tr>
<td>8</td>
<td>3</td>
<td>8</td>
<td>24</td>
<td>64</td>
<td>512</td>
<td>256</td>
</tr>
<tr>
<td>16</td>
<td>4</td>
<td>16</td>
<td>64</td>
<td>256</td>
<td>4,096</td>
<td>65,536</td>
</tr>
<tr>
<td>32</td>
<td>5</td>
<td>32</td>
<td>160</td>
<td>1,024</td>
<td>32,768</td>
<td>4,294,967,296</td>
</tr>
<tr>
<td>64</td>
<td>6</td>
<td>64</td>
<td>384</td>
<td>4,096</td>
<td>262,144</td>
<td>$1.84 \times 10^{19}$</td>
</tr>
<tr>
<td>128</td>
<td>7</td>
<td>128</td>
<td>896</td>
<td>16,384</td>
<td>2,097,152</td>
<td>$3.40 \times 10^{38}$</td>
</tr>
<tr>
<td>256</td>
<td>8</td>
<td>256</td>
<td>2,048</td>
<td>65,536</td>
<td>16,777,216</td>
<td>$1.15 \times 10^{77}$</td>
</tr>
<tr>
<td>512</td>
<td>9</td>
<td>512</td>
<td>4,608</td>
<td>262,144</td>
<td>134,217,728</td>
<td>$1.34 \times 10^{154}$</td>
</tr>
</tbody>
</table>
Comparing Running Times

**Comparing** running times

\[ T_A(n) \text{ is better than } T_B(n) \text{ if } T_A(n) \text{ grows slower than } T_B(n) \]
Running Times

**Constant**
- independent of input size

**Linear**
- depends on input size

**Quadratic**
- depends on square of input size
Running Times

- $O(1)$: independent of input size
- $O(n)$: depends on input size
- $O(n^2)$: depends on square of input size
- $O(n^3)$: depends on cube of input size
- $O(n^{70})$: depends on 70th power of input size
- $O(2^n)$: exponential in input size
Which Algorithm is Better?

- Algorithm A takes $T_A(n) = 30n + 10$ ops
- Algorithm B takes $T_B(n) = 5n$ ops
Which Algorithm is Better?

- Alg A takes $T_A(n) = 5n + 1000$ ops
- Alg B takes $T_B(n) = 10n + 2$ ops
- It depends on $n$

$rtime(A) < rtime(B) \iff 5n + 1000 < 10n + 2 \iff 5n > 998 \iff n > 199.6$
Which Algorithm is Better?

- Alg A takes $T_A(n) = 1000n^2$ ops
- Alg B takes $T_B(n) = n^8$ ops
- It depends on $n$

\[
\text{rtime}(A) < \text{rtime}(B) \iff 1000n^2 < n^8 \\
\iff 1000n^2 - n^8 < 0 \\
\iff n^2(1000 - n^6) < 0 \\
\iff 1000 - n^6 < 0 \\
\iff n > 1000^{1/6} \\
\iff n > 3.16…
\]
Comparing Running Times

Comparing \textit{asymptotic} running times

\[ T_A(n) \text{ is better than } T_B(n) \text{ if } \]

\textit{for large enough } \( n \)

\[ T_A(n) \text{ grows slower than } T_B(n) \]
Q: can we formalize all this mathematically?
Definition (Big-O): \( T_A(n) \) is \( O(T_B(n)) \) if there exists positive constants \( c \) and \( n_0 \) such that:

\[
T_A(n) \leq c \cdot T_B(n)
\]

for all \( n \geq n_0 \)

- \( T_A(n) \)'s order of growth is at most \( T_B(n) \)'s order of growth
- Examples
  - \( 2n+10 \) is \( O(n) \)
  - \( n^{10}+2019 \) is \( O(n^{10}) \) and also \( O(n^{50}) \)
Big-O

‣ How do we find “the Big-O of something”?
  ‣ Usually you “eyeball” it
  ‣ Then you try to prove it
    ‣ (though most of the time in CS16 it will be simple enough that you don’t need to prove it)
Eyeballing Big-O

- If \( T(n) \) is a polynomial of degree \( d \) then \( T(n) \) is \( O(n^d) \)
- In other words you can ignore
  - lower-order terms
  - constant factors
- Examples
  - \( 1000n^2+400n+739 \) is \( O(n^2) \)
  - \( n^{80}+43n^{72}+5n+1 \) is \( O(n^{80}) \)
- For Big-O, use the smallest upper bound
  - \( 2n \) is \( O(n^{50}) \) but that’s not really a useful bound
  - instead it is better to say that \( 2n \) is \( O(n) \)
Big-Omega

Definition (Big-$\Omega$): $T_A(n)$ is $\Omega(T_B(n))$ if there exists positive constants $c$ and $n_0$ such that:

$$T_A(n) \geq c \cdot T_B(n)$$

for all $n \geq n_0$

- $T_A(n)$’s growth rate is lower bounded by $T_B(n)$’s growth rate
- What about an upper and a lower bound?
  - We use Big-$\Theta$ notation
Eyeballing Big-Omega

- If $T(n)$ is a polynomial of degree $d$ then $T(n)$ is $\Omega(n^d)$
- In other words you can ignore
  - lower-order terms
  - constant factors
- Examples
  - $1000n^2+400n+739$ is $\Omega(n^2)$
  - $n^{80}+43n^{72}+5n+1$ is $\Omega(n^{80})$
- For the Big-$\Omega$, use the largest upper bound
  - $2n$ is $\Omega(\log n)$ but that’s not really a useful bound
  - instead it is better to say that $2n$ is $\Omega(n)$
Big-Theta

Definition (Big-$\Theta$): $T_A(n)$ is $\Theta(T_B(n))$ if it is $O(T_B(n))$ and $\Omega(T_B(n))$.

- $T_A(n)$’s growth rate is the same as $T_B(n)$’s
Eyeballing Big-Theta

- If $T(n)$ is a polynomial of degree $d$ then $T(n)$ is $\Theta(n^d)$
- In other words you can ignore
  - lower-order terms
  - constant factors
- Examples
  - $1000n^2+400n+739$ is $\Theta(n^2)$ since it is $O(n^2)$ and $\Omega(n^2)$
  - $n^{80}+43n^{72}+5n+1$ is $\Theta(n^{80})$ since it is $O(n^{80})$ and $\Omega(n^{80})$
Dynamic Programming
What is Dynamic Programming?

- Algorithm design paradigm/framework
  - Design efficient algorithms for optimization problems

- Optimization problems
  - “find the **best** solution to problem $X$”
  - “what is the **shortest** path between $u$ and $v$ in $G$”
  - “what is the **minimum** spanning tree in $G$”

- Can also be used for non-optimization problems
When is Dynamic Programming Applicable?

- Condition #1: sub-problems
  - The problem can be solved recursively
  - Can be solved by solving sub-problems

- Condition #2: overlapping sub-problems
  - Same sub-problems need to be solved many times
Sub-Problems

\[
\text{Sol}(\begin{array}{c}
\text{blue}\\
\text{green}\\
\text{red}\\
\text{yellow}
\end{array}) = \text{Sol}(\begin{array}{c}
\text{blue}\\
\text{green}
\end{array}) \oplus \text{Sol}(\begin{array}{c}
\text{red}\\
\text{yellow}
\end{array})
\]

\[
\text{Sol}(\begin{array}{c}
\text{blue}\\
\text{green}
\end{array}) = \text{Sol}(\begin{array}{c}
\text{blue}
\end{array}) \oplus \text{Sol}(\begin{array}{c}
\text{green}
\end{array})
\]

\[
\text{Sol}(\begin{array}{c}
\text{red}\\
\text{yellow}
\end{array}) = \text{Sol}(\begin{array}{c}
\text{red}
\end{array}) \oplus \text{Sol}(\begin{array}{c}
\text{yellow}
\end{array})
\]
Overlapping Sub-Problems

\[
\text{Sol}(\text{[blue, red, blue, red]}) = \text{Sol}(\text{[blue, red]}) \oplus \text{Sol}(\text{[blue, red]})
\]

\[
\text{Sol}(\text{[blue, red]}) = \text{Sol}(\text{[blue]}) \oplus \text{Sol}(\text{[red]})
\]

\[
\text{Sol}(\text{[blue, red]}) = \text{Sol}(\text{[red]}) \oplus \text{Sol}(\text{[blue]})
\]

Why solve red twice?
Why solve blue twice?
When is Dynamic Programming Applicable?

- Core idea
  - Decompose problem into its sub-problems
  - and if sub-problems are overlapping then
  - solve each sub-problem once and store the solution
  - use stored solution when you need to solve sub-problem again
Steps to Solving a Problem w/ DP

‣ What are the **sub-problems**?

‣ What is the “**magic**” step?
  ‣ Given solution to a sub-problem…
  ‣ …how do I **combine** them to get solution to the problem?

‣ Which **(topological) order** on sub-problems can I use?
  ‣ so that solutions to sub-problems available before I need them

‣ Design iterative **algorithm**
  ‣ that solves sub-problems in order and stores their solution
Fibonacci

0, 1, 1, 2, 3, 5, 8, 13, 21, 34, ...

Base cases:

\[ F(n) = F(n - 1) + F(n - 2) \]

Base cases:

\[ F(0) = 0 \text{ and } F(1) = 1 \]
Fibonacci (Dynamic Programming)

- Given **n** compute
  - \( \text{Fib}(n) = \text{Fib}(n-1) + \text{Fib}(n-2) \)
  - with base cases \( \text{Fib}(0) = 0 \) and \( \text{Fib}(1) = 1 \)

- What are the **sub-problems**?
  - \( \text{Fib}(n-1), \text{Fib}(n-2), \ldots, \text{Fib}(1), \text{Fib}(0) \)

- What is the **magic** step?
  - \( \text{Fib}(n) = \text{Fib}(n-1) + \text{Fib}(n-2) \)

Magic step is usually not provided!!
Fibonacci (Dynamic Programming)

- Which **topological order** should I use?
  - Fib(0), Fib(1), ..., Fib(n-1), Fib(n)
Fibonacci (Dynamic Programming)

- Design iterative algorithm

```python
function Fib(n):
    fibs = []
fibs[0] = 0
fibs[1] = 1

    for i from 2 to n:
        fibs[i] = fibs[i-1] + fibs[i-2]

    return fibs[n]
```
Fibonacci (Dynamic Programming)

```python
function Fib(n):
    fibs = []
    fibs[0] = 0
    fibs[1] = 1
    for i from 2 to n:
        fibs[i] = fibs[i-1] + fibs[i-2]
    return fibs[n]
```

- What’s the runtime of `Fib( )`?
  - Calculates Fibonacci numbers from 2 to `n`
    - Performs \( O(1) \) ops for each one
  - Runtime is \( O(n) \)