CMSC 132: OBJECT-ORIENTED PROGRAMMING II

Algorithmic Complexity II

Department of Computer Science
University of Maryland, College Park
Analyzing Algorithms

• **Goal**
  • Find asymptotic complexity of algorithm

• **Approach**
  • Ignore less frequently executed parts of algorithm
  • Find *critical section* of algorithm
  • Determine how many times critical section is executed as function of problem size
Critical Section of Algorithm

- Heart of algorithm
- Dominates overall execution time

Characteristics
- Operation central to functioning of program
- Usually contained inside deeply nested loops

Sources
- Loops
- Recursion
Computing Number of Iterations

- In the slides that follow we often need to compute how many times a loop is executed. Keep the following in mind:
  - for (int i = 1; i <= n; i++) { body }
    - Number of times body is executed: n - 1 + 1 → n
  - for (int i = 3; i <= n; i++) { body }
    - Number of times body is executed: n - 3 + 1 → n - 2
  - for (int i = 3; i < n; i++) { body } // condition is < not <=
    - Previous loop equivalent to
      - for (int i = 3; i <= n - 1; i++) { body }
        - Number of times body is executed: n - 1 - 3 + 1 → n - 3
  - for (int i = 0; i < n; i++) { body } // condition is < not <=
    - Number of times body is executed: n - 1 - 0 + 1 → n
Critical Section Example 1

• Suppose A, B, C are operations that do not involve loops (or recursive calls)
• Code (for input size n)

```
A
for (int i = 0; i < n; i++) {
    B
}
C
```

• Code execution
  • A \(\Rightarrow\) once
  • B \(\Rightarrow\) n times
  • C \(\Rightarrow\) once
• \(T(n) \Rightarrow 1 + n + 1 = O(n)\)
Critical Section Example 2

• Code (for input size \( n \))
  ```
  A
  for (int i = 0; i < n; i++) {
    B
    for (int j = 0; j < n; j++) {
      C
    }
  }
  ```
  D

• Code execution
  • \( A \) \( \Rightarrow \) once
  • \( B \) \( \Rightarrow \) \( n \) times
  • \( C \) \( \Rightarrow \) \( n^2 \) times
  • \( D \) \( \Rightarrow \) once
  • \( T(n) \) \( \Rightarrow \) \( 1 + n + n^2 + 1 = O(n^2) \)
Critical Section Example 3

• Code (for input size $n$)
  
  A
  
  for (int $i = 0$; $i < n$; $i++$) {
    for (int $j = i + 1$; $j < n$; $j++$) {
      B
    }
  }

• Code execution
  
  • A $\Rightarrow$ once
  
  • B $\Rightarrow$ $(n-1)+(n-2)+(n-3)+…+ 3+2+1+0 = \frac{1}{2} n(n-1)$ times

• $T(n) \Rightarrow 1 + \frac{1}{2} n^2 - \frac{1}{2} n = O(n^2)$
Critical Section Example 4

• Code (for input size \( n \))

```java
A
for (int i = 0; i < n; i++) {
    for (int j = 0; j < 10000; j++) {
        B
    }
}
```

• Code execution
  • \( A \) \( \Rightarrow \) once
  • \( B \) \( \Rightarrow \) 10000\( n \) times

• \( T(n) \) \( \Rightarrow \) 1 + 10000\( n \) = \( O(n) \)
• Just because we have nested loops we don’t necessarily have \( O(n^2) \)
Critical Section Example 5

- Code (for input size $n$)

```java
for (int i = 0; i < n/2; i++)
    for (int j = 0; j < n/2; j++)
        A

for (int i = 0; i < n; i++)
    for (int j = 0; j < n; j++)
        B
```

- Code execution
  - $A \Rightarrow n^2/4$ times
  - $B \Rightarrow n^2$ times
  - $T(n) \Rightarrow n^2/4 + n^2 = O(n^2)$

- You can have more than one critical section
Critical Section Example 6

- Code (for input size $n$)

  ```
  i = 1
  while (i < n) {
    A
    i = 2 \times i
  }
  B
  ```

- Code execution
  - $i = 1 \Rightarrow 1$ time
  - $A \Rightarrow \log(n)$ times
  - $i = 2 \times i \Rightarrow \log(n)$ times
  - $B \Rightarrow 1$ time
  - $T(n) \Rightarrow 1 + \log(n) + \log(n) + 1 = O(\log(n))$

  Use a trace table to analyze the number of times body is executed
Asymptotic Complexity of Recursive Algorithms

- How can we compute the complexity of recursive algorithms?
  - By using a recurrence relation
    - Example: $T(n) = 2 \cdot T(n/2) + O(n)$, $T(1) = O(1)$
- In a recurrence relation $T(\cdot)$ appears on both sides of the $=$ sign
- By solving the recurrence relation, we can determine the algorithmic complexity of the algorithm described by the relation
- When you write a recurrence relation you write two equations:
  - One for the **base case**
  - One for the **general case**
- About the base case equation
  - Often an $O(1)$ operation
  - Base case involves input of size one, so $T(1) = O(1)$
  - You can also have base case of size zero, so $T(0) = O(1)$
- Solving the recurrence relation
  - You can use induction
  - We can solve it following a non-formal approach where you identify a pattern
Example: Mergesort

- Recursively sort the first half
- Recursively sort the second half
- Merge the two halves to get the array sorted

Mergesort overview
Recursion: Mergesort

- MergeSort

\[ \text{Mergesort}(\text{Array of size } n) \{ \]
  \[
  \quad \text{if } (n == 1) \{ \\
  \quad \quad \text{return;} \\
  \quad \} \text{ else } \{ \\
  \quad \quad \text{Mergesort}(\text{First } n/2 \text{ elements in the array}) \\
  \quad \quad \text{Mergesort}(\text{Last } n/2 \text{ elements in the array}) \\
  \quad \quad \text{Merge the two halves} \]

\[
\} \}
\]

- Base case: \( T(1) = 1 \)
- General case (recurrence relation): \( T(n) = 2 \ T(n/2) + n \)
Solving Recurrence Relation (Mergesort)

- **Base case:** $T(1) = 1$
- **General case (recurrence relation):** $T(n) = 2 \cdot T(n/2) + n$
- Solving recurrence relation
  - $T(n) = 2 \cdot T(n/2) + n$
    - $= 2 \cdot (2 \cdot T(n/4) + n/2) + n$
    - $= 4 \cdot T(n/4) + 2n$
    - $= 4 \cdot (2 \cdot T(n/8) + n/4) + 2n$
    - $= 8 \cdot T(n/8) + 3n$
    - $= 16 \cdot T(n/16) + 4n$
    - … at this point you can see a pattern …
    - $= 2^k \cdot T(n/2^k) + kn$

- $T(1) = 1$ will allow us to end the derivation above. What is the value of $k$ for $n/2^k$ be equal to 1?

  \[
  \frac{n}{2^k} = 1 \implies k = \log_2 n
  \]

- Replacing $k$ with $\log_2 n$
  - $T(n) = 2^k \cdot T(n/2^k) + kn$
    - $= 2^{\log_2 n} \cdot T(1) + (\log_2 n) \cdot n$
    - $T(n) = n + n \log_2 n$
    - $= O(n \log(n))$
Recurrence Relations

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Recurrence Relation</th>
<th>Big-O</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sequential Search</td>
<td>$T(n) = T(n-1) + O(1)$</td>
<td>$O(n)$</td>
</tr>
<tr>
<td>Binary Search</td>
<td>$T(n) = T(n/2) + O(1)$</td>
<td>$O(\log n)$</td>
</tr>
<tr>
<td>Tree Traversal</td>
<td>$2 \times T(n/2) + O(1)$</td>
<td>$O(n)$</td>
</tr>
<tr>
<td>Mergesort</td>
<td>$T(n) = 2 \times T(n/2) + O(n)$</td>
<td>$O(n\log(n))$</td>
</tr>
</tbody>
</table>
Comparing Complexity

- Compare two algorithms
  - \( f(n) \), \( g(n) \)
- Determine which increases at faster rate
  - As problem size \( n \) increases
- Can compare ratio
  - \( \lim_{n \to \infty} \frac{f(n)}{g(n)} \)
    - If \( \infty \), \( f() \) is larger
    - If 0, \( g() \) is larger
    - If constant, same complexity
- Example (\( \log(n) \) vs. \( n^{1/2} \))

\[
\lim_{n \to \infty} \frac{f(n)}{g(n)} \quad \lim_{n \to \infty} \frac{\log(n)}{n^{1/2}} \quad 0
\]
Additional Complexity Measures

• Upper bound
  • Big-O \( \Rightarrow O(\ldots) \)
  • Represents upper bound on # steps

• Lower bound
  • Big-Omega \( \Rightarrow \Omega(\ldots) \)
  • Represents lower bound on # steps
2D Matrix Multiplication Example

- Problem
  - $C = A \times B$

- Lower bound (best case)
  - $\Omega(n^2)$
    - Required to examine 2D matrix

- Upper bounds
  - $O(n^3)$
    - Basic algorithm
  - $O(n^{2.807})$
    - Strassen’s algorithm (1969)
  - $O(n^{2.376})$
    - Coppersmith & Winograd (1987)

- Improvements still possible (open problem)
  - Since upper & lower bounds do not match
Resources

- [http://bigocheatsheet.com/]