CMSC 132: OBJECT-ORIENTED PROGRAMMING II

Algorithmic Complexity II

Department of Computer Science
University of Maryland, College Park

© Department of Computer Science UMD

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

© Department of Computer Science UMD

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

© Department of Computer Science UMD

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

© Department of Computer Science UMD

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
```

critical section

- Code execution
  - A ⇒ once
  - B ⇒ n times
  - C ⇒ 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
```

critical section

- Code execution
  - A $\Rightarrow$ once
  - B $\Rightarrow$ n times
  - C $\Rightarrow$ n^2 times
  - D $\Rightarrow$ once
- $T(n) \Rightarrow 1 + n + n^2 + 1 = O(n^2)$

© Department of Computer Science UMD

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
    }
}
```

critical section

- Code execution
  - $A \Rightarrow$ once
  - $B \Rightarrow (n-1)+(n-2)+(n-3)+\ldots+3+2+1+0 = \frac{1}{2}\, n(n-1)$ times
- $T(n) \Rightarrow 1 + \frac{1}{2} n^2 - \frac{1}{2} n = O(n^2)$

© Department of Computer Science UMD

Critical Section Example 4

- Code (for input size n)

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

critical section

- Code execution
  - A $\Rightarrow$ once
  - B $\Rightarrow$ 10000n times
- T(n) $\Rightarrow$ 1 + 10000n = O(n)
- Just because we have nested loops we don't necessarily have $O(n^2)$

© Department of Computer Science UMD

Critical Section Example 5

- Code (for input size n)

```
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
```

critical sections

- 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

© Department of Computer Science UMD

Critical Section Example 6

- Code (for input size n)

  i = 1
  while (i < n) {
      A
      i = 2 × i
  }
  B

critical section

- Code execution
  - i = 1 ⇒ 1 time
  - A ⇒ log(n) times
  - i = 2 x i ⇒ log(n) times
  - B ⇒ 1 time
- T(n) ⇒ 1 + log(n) + log(n) + 1 = O(log(n))
- Use a trace table to analyze the number of times body is executed

© Department of Computer Science UMD

Asymptotic Complexity of Recursive Algorithms

- How can we compute the complexity of recursive algorithms?
  - By using a recurrence relation
    - Example: T(n) = 2 T(n/2) + O(n), T(1) = O(1)
- In a recurrence relation T(..) 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
- Reference: https://users.cs.duke.edu/~ola/ap/recurrence.html

© Department of Computer Science UMD

Example: Mergesort

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

input	M	E	R	G	E	S	O	R	T	E	X	A	M	P	L	E
sort left half	E	E	G	M	O	R	R	S	T	E	X	A	M	P	L	E
sort right half	E	E	G	M	O	R	R	S	A	E	E	L	M	P	T	X
merge results	A	E	E	E	E	G	L	M	M	O	P	R	R	S	T	X

Mergesort overview

© Department of Computer Science UMD

Recursion : Mergesort

- MergeSort

```
Mergesort(Array of size n) {
    if (n == 1) {
        return;
    } else {
        Mergesort(First n/2 elements in the array)
        Mergesort(Last n/2 elements in the array)
        Merge the two halves          n operations
    }
}
```

- Base case: T(1) = 1
- General case (recurrence relation): T(n) = 2 T(n/2) + n

© Department of Computer Science UMD

Solving Recurrence Relation (Mergesort)

- **Base case:** $T(1) = 1$
- **General case (recurrence relation):** $T(n) = 2\,T(n/2) + n$
- Solving recurrence relation
  - $T(n) = 2\,T(n/2) + n$

    $= 2\,(2\,T(n/4) + n/2) + n$

    $= 4\,T(n/4) + 2n$

    $= 4\,(2\,T(n/8) + n/4) + 2n$

    $= 8\,T(n/8) + 3n$

    $= 16\,T(n/16) + 4n$

    … at this point you can see a pattern …

    $= 2^k T(n/2^k) + kn$

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

    $$n/2^k = 1 \rightarrow k = \log_2 n$$

  - Replacing k with $\log_2 n$
  - $T(n) = 2^k T(n/2^k) + kn$

    $= 2^{\log_2 n}\, T(1) + (\log_2 n)\, n$

$$T(n) = n + n\log_2 n$$
$$= O(n\log(n))$$

© Department of Computer Science UMD

Recurrence Relations

Recurrence Relations

Algorithm	Recurrence Relation	Big-O
Sequential Search	T(n) = T(n-1) + O(1)	O(n)
Binary Search	T(n) = T(n/2) + O(1)	O(log n)
Tree Traversal	2 T(n/2) + O(1)	O(n)
Mergesort	T(n) = 2 T(n/2) + O(n)	O(nlog(n))

© Department of Computer Science UMD

Comparing Complexity

- Compare two algorithms
  - f(n), g(n)
- Determine which increases at faster rate
  - As problem size n increases
- Can compare ratio
  - If ∞, f() is larger
  - If 0, g() is larger
  - If constant, same complexity
- Example (log(n) vs. n½)

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

$$\lim_{n \to \infty} \frac{f(n)}{g(n)} \rightarrow \lim_{n \to \infty} \frac{\log(n)}{n^{1/2}} \rightarrow 0$$

© Department of Computer Science UMD

Additional Complexity Measures

- Upper bound
  - Big-O ⇒ $O(\ldots)$
  - Represents upper bound on # steps
- Lower bound
  - Big-Omega ⇒ $\Omega(\ldots)$
  - Represents lower bound on # steps

© Department of Computer Science UMD

2D Matrix Multiplication Example

- Problem
  - C = A * 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

© Department of Computer Science UMD

Resources

- http://bigocheatsheet.com/