CS 113
DISCRETE STRUCTURES

Chapter 3: Algorithms

HOMEWORK

- **All homework is from the Exercises**
  - **No problems are from the Review Exercises**
- **Section 3.2 (p. 127): 19, 21**
- **Section 3.3 (p. 131): 1-5 (all), 13, 22-24 (all)**
- **Section 3.4 (p. 137): 8, 9, 12, 27, 28**
- **Section 3.5 (p. 149): 1-12 (all), 17, 19**
- **Section 3.7 (p. 160): 1-9 (all)**

THE SPEED OF AN ALGORITHM

- **You are writing a program**
- **You want to sort the data in an array**
- **There are many sort algorithms to choose from**
- **You want to know which algorithm is fastest**
- **How can you compare algorithms?**

TRYING TO COMPARE THE ALGORITHMS

- **We could set a timer before the algorithm starts, and then print the elapsed time when the algorithm finishes**
- **This can be unreliable**
  - **For example, our program can be competing with another program, which will slow down our program**
  - **In a typical computer, there are many background programs**

COMPARING ALGORITHMS AS ALGORITHMS

- There are actually two things we could measure
- We could measure how much time the algorithm requires
- We call this time complexity
- We could measure how much memory the algorithm requires
- We call this space complexity
- Measuring complexity (either type) is called analysis of algorithms

PROBLEM SIZE

- **Both of these measures are based on problem size**
  - **Problem size refers to the number of items in the problem**
  - **In this case, the problem size is how many numbers we want to add**
  - **Similarly, the problem size for the sort would be the number of items we need to sort**
- **Since timers are unreliable, we will do this by counting operations**

COMPARING THREE SPECIFIC ALGORITHMS

- **Suppose we want to add the numbers from 1 to n**
- **We can do this three different ways**
  1. **We can just add all the numbers**
  2. **We can add 1 many times**
  3. **We can use the formula from math**
- **Again, we want to compare the three algorithms**

ALGORITHM A

```
long sum = 0;
for (int i = 1;  i<= n;  i++)
   sum = sum + i;
```

ALGORITHM B

```
long sum = 0;
for (int i=1;  i<= n; i++)
   for (int j=1;  j<=i;  j++)
      sum = sum + 1;
```

ALGORITHM C

```
long sum = n * (n+1) / 2;
```

BASIC OPERATIONS

- **A basic operation is the one that takes the most time**
  - **In Algorithm A, that's addition**
  - **In Algorithm B, that's also addition**
  - **In Algorithm C, that's addition, multiplication, and division**
- **We focus on the basic operations**

- **Notice that all these are math operations**
- **We often also consider comparisons**
  - **They do take some time, but generally not as much time as math**

CALCULATING THE TIME COMPLEXITY OF ALGORITHM A

- **There are two parts to focus on**
  - **The loop**
  - **The other code**

TIME COMPLEXITY OF THE LOOP ITSELF

- **Timing the code requires accounting for loop overhead**
- **For the loop itself, there are**
  - **1 assignment to i**

    i = 1
  - **n+1 comparisons**

    i <= n (This is hidden in the for loop)
  - **n additions**

    i = i + 1 (These additions are also hidden in the for loop)
  - **n assignments**

    Again, i = i + 1 (These assignments are also hidden in the for loop)

TIME COMPLEXITY OF THE CODE ITSELF

- **1 assignment**

  sum = 0
- **n additions**

  sum = sum + i (In the loop)
- **n more assignments**

  sum = sum + i (Again, in the loop)

TOTAL TIME COMPLEXITY

- **Total assignments**

  1 + n + 1 + n = 2n + 2
- **Total additions**

  n + n = 2n
- **Total comparisons**

  n + 1
- **These three operations could require different amounts of time, but let's suppose all three are equally slow**
- **Then we have**

  (2n + 2) + (2n) + (n + 1) operations, or 5n + 3 operations

THE COMPLEXITY OF THE CODE

- **So, if the problem size is n, it takes 5n + 3 operations**
- **If we add one item to the problem**
  - **The problem size changes to n+1**
  - **The time changes to 5(n+1) + 3**
  - **That is 5n + 5 + 3, or (5n + 3) + 5**
  - **This is exactly 5 more time units than the original**
- **Each time we add one more item to the problem, it adds 5 time units**
- **Adding 4 to the problem size adds 20 time units**

ALGORITHM B

- There are essentially three parts to calculating the time
  - The overhead for the outer loop (the "i" loop)
  - The overhead for the inner loop (the "j" loop)
  - The time for the actual calculations of the sum

ALGORITHM B

- The outer loop is performed n times
  - The overhead is 3n + 2 as in Algorithm A
- The inner loop is performed a varying number of times, based on i in the outer loop
  - I will use the table on the next slide to find the overhead
- The calculation of the sum is performed n (n+1) / 2 times in the loop
  - There Is one addition and one assignment, giving n (n+1) operations

ALGORITHM B-INNER LOOP OVERHEAD

		j=1	j<=i	j++
i = 1	First iteration	1	2	1
i = 2	Second iteration	1	3	2
…				
i = n	nth iteration	1	n+1	n
Totals (Sum)		n	n(n+3)/2	n(n+1)/2

TIME COMPLEXITY OF THE CODE ITSELF

- **So, in total there are $2n^2 + 7n + 2$ operations**
- **Here, adding 1 to the problem size adds $4n + 9$ to the required amount of time units**
  - **Here n is the problem size before you add 1**

COMPARING WITH ALGORITHM A

- **This is not like Algorithm A!**
  - **If n = 3, adding 1 item adds 21 time units to the time required**
  - **If n = 12, adding 1 item adds 57 time units to the time required**
- **The amount of time required for each increase of 1 item in the problem size keeps growing**

ALGORITHM C

- Algorithm C requires
  - 1 addition
  - 1 multiplication
  - 1 division
- Algorithm C has 3 operations
- Independent of what n is, adding 1 to the problem size has no effect on the time

GROWTH RATES

- When n is small, the time difference between algorithms is usually minor
  - In that case, choose the algorithm you like
- When n is large, a bad algorithm can waste a lot of time
- So, we assume n is large when discussing growth rates

ALGORITHM ANALYSIS

- **What we are talking about is called analysis of algorithms**
- **The calculations in this chapter involve the calculus idea of a limit**
- **This is called asymptotic analysis**

CONSTANT MULTIPLIERS

- **We ignore constant coefficients in these formulas because we are computing orders, not actual times**
  - **If the numbers were taken literally**
    - **Changing time units would change constants**
    - **Changing computers would change constants**
    - **Changing the programming language would change constants**

BIG O NOTATION

- So we will say Algorithm A is O (n)
  - The actual function is 5n + 3, but if n is large, adding 3 to 5n is not noticeable
  - We also factor out (ignore) the coefficient 5 for the reasons mentioned above
- Using the same reasoning, we say Algorithm B is O (n^2)
- And also, Algorithm C is constant, independent of n, or O(1)

EXPLAINING BIG O

- Growth rates are O (function of n)
  - Here O stands for the word "order"
  - For example, we could write $O(n^2)$
    - The function inside the parentheses means that the algorithm grows no faster than the function
  - We choose the best function we can
    - For example, we could use $O(n^3)$ instead of $O(n^2)$, but this is not useful (not the best)
  - We read $O(n^2)$ as "Big O of n^2"

SOME POSSIBILITIES FOR GROWTH RATES

- **O(1)**
  - **(constant)**
- **log(log n)**
- **log n**
  - **The base of the log doesn't matter**

- **n**
- **n log n**
- **n^2**
- **n^3**
- **2^n**
- **n!**

BIG Θ NOTATION

- **Two algorithms are said to be Big θ of each other if their Big O time requirements are constant multiples of each other**
- **This means their growth rates are essentially the same**

USING BIG Θ NOTATION

- So, a more precise way of stating the rates of growth is
  - Algorithm A is $\theta(n)$
  - Algorithm B is $\theta(n^2)$
  - Algorithm C is $\theta(1)$

COMBINING TIMES-PART 1

- **If there are several algorithms one after another, the time complexity is**
  - **the largest individual time complexity**
  - **or the sum, if they are the same**

COMBINING TIMES-PART 2

- **If there is decision logic to choose between algorithms, the time complexity is**
  - **the largest individual time complexity**
  - **plus time required for the decision**

COMBINING TIMES-PART 3

- **If a loop contains an algorithm whose growth rate is g(n), then its complexity is**
  - **number of loop iterations x g(n)**
  - **plus, again, time required for the loop overhead**

MEASURING TIMES IN PRACTICE

- It often happens that there are different functions for different “versions” of the same problem
- For example, if you have to sort an array, you could
  - Sort an already sorted array
  - Sort a very scrambled array
  - Sort a somewhat scrambled array
- For that reason, we usually find different times for the algorithm
- We call them the best, worst, and the average times

HOMEWORK

- **Section 3.5 (p. 149): 1-12 (all), 17, 19**

QUESTIONS

- Any questions?