Week 11 - Wednesday

COMP 2230

Last time

- Graphing functions
- Big-O, Big-Omega, and Big-Theta notation

Questions?

Assignment 5

Logical warmup

- You have three pitchers whose capacities are three, four, and seven quarts
- Only the seven-quart pitcher is full
- Pour the fewest times to separate the water into two-, two-, and three-quart quantities

Asymptotic Notations Continued

Orders of functions

- If $1 < x$, then
  - $x < x^2$
  - $x^2 < x^3$
  - …
- So, for $r, s \in \mathbb{R}$, where $r < s$ and $x > 1$,
  - $x^r < x^s$
  - By extension, $x^r \; is \; O(x^s)$

Proving bounds

- Prove a Θ bound for $g(x) = \frac{1}{4}(x - 1)(x + 1)$ for $x \in \mathbb{R}$
- Prove that x^2 is not $O(x)$
  - Hint: Proof by contradiction

Polynomials

- Let $f(x)$ be a polynomial with degree n
  - $f(x) = a_n x^n + a_{n-1} x^{n-1} + a_{n-2} x^{n-2} \ldots + a_1 x + a_0$
- By extension from the previous results, if a_n is a positive real, then
  - $f(x)$ is $O(x^s)$ for all integers $s \geq n$
  - $f(x)$ is $\Omega(x^r)$ for all integers $r \leq n$
  - $f(x)$ *is* $\Theta(x^n)$
- Furthermore, let $g(x)$ be a polynomial with degree m
  - $g(x) = b_m x^m + b_{m-1} x^{m-1} + b_{m-2} x^{m-2} \ldots + b_1 x + b_0$
- If a_n and b_m are positive reals, then
  - $f(x)/g(x)$ is $O(x^c)$ for real numbers $c > n - m$
  - $f(x)/g(x)$ is **not** $O(x^c)$ for real numbers $c < n - m$
  - $f(x)/g(x)$ is $\Theta(x^{n-m})$

Algorithm Efficiency and Exponential and Logarithmic Functions

Three-Sentence Summary

Algorithm Efficiency

Extending notation to algorithms

- We can easily extend our Ω-, O-, and Θ- notations to analyzing the running time of algorithms
- Imagine that an algorithm A is composed of some number of elementary operations (usually arithmetic, storing variables, etc.)
- We can imagine that the running time is tied purely to the number of operations
- This is, of course, a lie
  - Not all operations take the same amount of time
  - Even the **same** operation takes different amounts of time depending on caching, disk access, etc.

Running time

- First, assume that the number of operations performed by A on input size n is dependent only on n, not the values of the data
  - If $f(n)$ is $\Theta(g(n))$, we say that A **is** $\Theta(g(n))$ or that A **is of order** $g(n)$
- If the number of operations depends not only on n but also on the values of the data
  - Let $b(n)$ be the **minimum** number of operations where $b(n)$ is $\Theta(g(n))$, then we say that **in the best case**, A **is** $\Theta(g(n))$ or that A **has a best case order of** $g(n)$
  - Let $w(n)$ be the **maximum** number of operations where $w(n)$ is $\Theta(g(n))$, then we say that **in the worst case**, A **is** $\Theta(g(n))$ or that **A has a worst case order of** $g(n)$

Time

Approximate Time for $f(n)$ Operations Assuming One Operation Per Nanosecond				
$f(n)$	$n = 10$	$n = 1{,}000$	$n = 100{,}000$	$n = 10{,}000{,}000$
$\log_2 n$	3.3×10^{-9} s	10^{-8} s	1.7×10^{-8} s	2.3×10^{-8} s
n	10^{-8} s	10^{-6} s	0.0001 s	0.01 s
$n \log_2 n$	3.3×10^{-8} s	10^{-5} s	0.0017 s	0.23 s
n^2	10^{-7} s	0.001 s	10 s	27.8 hours
n^3	10^{-6} s	1 s	11.6 days	31,668 years
2^n	10^{-6} s	3.4×10^{284} years	3.1×10^{30086} years	2.9×10^{3010283} years

Computing running time

- With a single `for` (or other) loop, we simply count the number of operations that must be performed:

```
int p = 0;
int x = 2;
for (int i = 2; i <= n; ++i) {
     p = (p + i)*x;
}
```

- Counting multiplies and adds, $n - 1$ iterations times 2 operations $= 2n - 2$
- As a polynomial, $2n - 2$ is $\Theta(n)$

Nested loops

- When loops do not depend on each other, we can simply multiply their iterations (and asymptotic bounds)

```
int p = 0;
for (int i = 2; i <= n; ++i) {
     for (int j = 3; j <= n; ++j) {
          ++p;
     }
}
```

- Clearly $(n - 1)(n - 2)$ is $\Theta(n^2)$

Trickier nested loops

- When loops depend on each other, we have to do more analysis
  What's the running time here?

```
int s = 0;
for (int i = 1; i <= n; ++i) {
     for (int j = 1; j <= i; ++j) {
          s = s + j*(i - j + 1);
     }
}
```

- Arithmetic sequence saves the day (for the millionth time)

Iterations with floor

- When loops depend on floor, what happens to the running time?

```
int a = 0;
for (int i = n/2; i <= n; ++i) {
     a = n - i;
}
```

- Floor is used implicitly here, because we're using integer division
- What's the running time? Hint: Consider n as odd or as even separately

Sequential search

- Consider a basic sequential search algorithm:

```
int search(int[] array, int n, int value) {
      for (int i = 0; i < n; ++i) {
            if (array[i] == value) {
                  return i;
            }
      }
      return -1;
}
```

- What's its best case running time?
- What's its worst case running time?
- What's its average case running time?

Insertion sort algorithm

- Insertion sort is a common introductory sort
- It's suboptimal, but it's one of the fastest ways to sort a small list (10 elements or fewer)
- The idea is to sort initial segments of an array, insert new elements in the right place as they are found
- So, for each new item, keep moving it up until the element above it is too small (or we hit the top)

Insertion sort in code

```
public static void sort( int[] array, int n) {
  for (int i = 1; i < n; ++i) {
     int next = array[i];
     int j = i - 1;
     while (j != 0 && array[j] > next) {
          array[j+1] = array[j];
          --j;
     }
     array[j] = next;
  }
}
```

Best case analysis of insertion sort

- What is the best case analysis of insertion sort?
- Hint: Imagine the array is already sorted

Worst case analysis of insertion sort

- What is the worst case analysis of insertion sort?
- Hint: Imagine the array is sorted in reverse order

Average case analysis of insertion sort

- What's the average case analysis of insertion sort?
- Much harder than the previous two!
- Let's look at it recursively
- Let E_k be the average number of comparisons needed to sort k elements
- E_k can be computed as the sum of the average number of comparisons needed to sort $k - 1$ elements plus the average number of comparisons (x) needed to insert the k^{th} element in the right place
  - $E_k = E_{k-1} + x$

Finding x

- We can employ the idea of **expected value** from probability
- There are k possible locations for the element to go
- We assume that any of these k locations is equally likely
- For each turn of the loop, there are 2 comparisons to do
- There could be 1, 2, 3, … up to k turns of the loop
- Thus, weighting each possible number of iterations evenly gives us

$$x = \sum_{j=1}^{k} \frac{1}{k} 2j = \frac{2}{k}\left(\frac{k(k+1)}{2}\right) = k + 1$$

Finishing the analysis

- Having found x, our recurrence relation is:
- $E_k = E_{k-1} + k + 1$
- Sorting one element takes no time, so $E_1 = 0$
- Solve this recurrence relation!
- Well, if you really banged away at it, you might find:
  - $E_n = \frac{1}{2}(n^2 + 3n - 4)$
- By the polynomial rules, this is $\Theta(n^2)$ and so the average case running time is the same as the worst case

Upcoming

Next time…

- Exponential and logarithmic functions
- Review for Exam 3

Reminders

- **Finish Assignment 5**
- Study for Exam 3 on Monday