CS 310: Computational Complexity

Connor Baker

January 24, 2019

Warm-up Exercise

- Let's make a box that will hold anything
- Let's make a bunch of boxes
- Let's play with the boxes
- Let's add `JavaDoc` comments

Algorithm Analysis

- An algorithm is how you do things
  - Data structures need algorithms to perform operations on the data they contain
- Algorithm analysis analyzes how well an algorithm performs
  - The definition of well depends on the problem

Three Complexities

- Time complexity
  - How much time given a lot of data?
- Space complexity
  - How much memory given a lot of data?
- Implementation complexity
  - How hard is the algorithm to write?

Mathematical Analysis

- Decompose the program into individual operations
- Each operation takes some constant amount of time and is execute with some frequency
  - The exact time depends on the machine and compiler
  - The frequency depends on the algorithm and input
- Notes:
  - Can be used to predict performance
  - Machine independent for comparison purpose
  - May not be easy to analyze

Modeling Time/Space Complexity

- Represent complexity as a function of the input
- When referring to the time complexity, we use $T(n)$ where n is usually the **input size**.

Sample Problems

- Let's look at a simple program and compute $T(n)$. Assume that each statement takes some constant time t to execute.

```java
//input array A is of size n
void method1(int [] A) {
    int n = A.length; return A[0]+n;
}
```

  - Takes time 2 to execute

- What about $T(n)$ of a single loop?

```java
void method2(int[] A, int m) {
    int n = A.length;
    for(int i = 0; i < n; i++) {
        A[i] += m;
    }
}
```

  - Takes time $3n + 2$ to execute

- What if there are more operations in that loop?

```java
void method3(int[] A, int m) {
    int n = A.length;
    for(int i = 0; i < n; i++) {
        A[i] += m;
        System.out.println(A[i]);
    }
}
```

  - Takes time $4n + 2$ to execute

- What if we have sequential loops?

```java
void method4(int[] A, int m) {
    int n = A.length;
    for(int i = 0; i < n; i++) {
        A[i] += m;
    }
    for(int i = 0; i < n; i++) {
        System.out.println(A[i]);
    }
}
```

  - Takes time $6n + 3$ to execute

- What if there's more than one factor?

```java
void method5(int[] A, int[] B) {
    int n = A.length;
    int m = B.length;
    for(int i = 0; i < n; i++) {
        System.out.println(A[i]);
    }
    for(int i = 0; i < m; i++) {
        System.out.println(B[i]);
    }
}
```

  - Takes time $6n + 4$ to execute

- What if we have nested loops?

```java
void method6(int[] A) {
    int n = A.length;
    for(int i = 0; i < n; i++) {
        for(int j = 0; j < n; j++) {
            System.out.println("("+A[i]+","+A[j]+")");
        }
    }
}
```

  - Takes time $2 + 2n + n + 3n^2 = 3n^2 + 3n + 2$ to execute

- What if we have multi-factor nested loops?

```java
void method7(int[] A, int[] B) {
    int n = A.length;
    int m = B.length;
    for(int i = 0; i < n; i++) {
        for(int j = 0; j < m; j++) {
            System.out.println("("+A[i]+","+B[j]+")");
        }
    }
}
```

  - Takes time $3 + 2n + n + 2m + nm = nm + 2(n + m) + n + 3$ to execute

Algorithm Time Complexity

- Algorithmic time/space complexity depend on problem size
- Often have some input parameter like n that is representative of the problem size
- Our previous examples have as input an array of size n
- Model time/space complexity as functions of those parameters

Big O Notation

- **Big-O** notation: bounding how fast functions grow based on input/problem size
- $T(n)$ is $O(F(n))$ if there are positive constants c and n_0 such that

$$n \geq n_0,\ T(n) \leq cF(n)$$

- Bottomline:
  - If $T(n)$ is $O(F(n))$, then $F(n)$ grows as fast or faster than $T(n)$

Big-O Example

- Show that $f(n) = 2n^2 + 3n + 2$ is $O(n^3)$
- Pick $c = 0.5$
  - For $n_0 = 6,\ 0.5 \cdot n^3 \geq 2n^2 + 3n + 2$
- Exercise:
  - Can you show that $g(n) = n^3$ is $O(2n^2 + 3n + 2)$?
    * Of course! Pick some arbitrarily big n_0, or find the least value that does the job by setting them equal to each other and solving for the intersection.

Basic Rules of Big-O

- Constant additions disappear
- Constant multiples disappear
- Non-constant multiples multiply
  - Doing a constant operation $2N$ times is $O(N)$
  - Doing a $O(N)$ operation $N/2$ times is $O(N^2)$
  - Need space for half an array with N elements is $O(N)$ space overhead
- Function calls are not free (including library calls, though they are usually very well optimized)

Growth Ordering

Name	Leading Term	Big-O	Example
Constant	$1, 5, c$	$O(1)$	$2.5, 85, 2c$
Log-Log	$\log(\log(n))$	$O(\log(\log(n))$	$10 + (\log(\log(n) + 5)$
Log	$\log(n)$	$O(\log(n))$	$5\log(n) + 2\log(n^2)$
Linear	n	$O(n)$	$10n + \log(n)$
N-log-N	$n\log(n)$	$O(n\log(n))$	$3.5n\log(n) + 10n + 8$
Super-linear	$n^{1.x}$	$O(n^{1.x})$	$2n^{1.2} + 3n\log(n) - n + 2$
Quadratic	n^2	$O(n^2)$	$n^2 + n\log(n)$
Cubic	n^3	$O(n^3)$	$0.1n^3 + 8n^{1.5} + \log(n)$
Polynomial	n^c	$O(n^c)$	$a_n x^n + \cdots + a_1 x + a_0$
Exponential	c^n	$O(c^n)$	$8(2^n) - n + 2$
Factorial	$n!$	$O(n!)$	$0.25n! + 10n^{100} + 2n^2$

Quick Rules of Thumb

- $O(1)$ - usually doing something that takes a fixed amount of time regardless of the problem/input size, no matter how long that time is
- $O(\log(n))$ - dividing a problem in half repeatedly and working on only one half each time
- $O(n)$ - doing something with each item of data (or a fraction of the data, like $n/2$)
- $O(n\log(n))$ - dividing a problem in half repeatedly and working on both halves each time
- $O(n^2)$ - nested loops that both go through each item
- $O(n^3)$ - three nested loops that each go through all data
- $O($[anything more than n^x]$)$ - you're usually doing it wrong

Common Patterns

- Adjacent loops are additive: $2 \times n$ is $O(n)$
- Nested loops are multiplicative (usually a polynomial)
- Repeated halving usually involves a logarithm
  - Binary search is $O(\log(n))$
  - Fastest sorting algorithms are $O(n \log(n))$
  - Proofs are harder because they generally require solving recurrence relations
- There are lots of special cases – so be careful!

Other Bounding Notations

- Big-O: Upper bound
  - $2n^2 + 3n + 2$ is $O(n^3)$ and $O(2^n)$ and $O(n^2)$
- Big-Ω: Lower bound
  - $T(n)$ is $\Omega(F(n))$ if there are positive constants c and n_0 such that when $n \geq n_0,\ T(n) \geq cF(n)$
  - $2n^2 + 3n + 2$ is $\Omega(n)$ and $\Omega(\log(n))$ and $\Omega(n^2)$
- Big-Θ: Upper and lower bound
  - If something is $O(F(n))$ and $\Omega(F(n))$ it is $\Theta(F(n))$
  - $2n^2 + 3n + 2$ is $\Theta(n^2)$
- Little-o: Upper bounded by but not lower bounded by
  - $T(n)$ grows much slower than $F(n)$
  - $2n^2 + 3n + 2$ is $o(n^3)$

Worst, Average, or Best Case?

- Plan for the worst, hope for the best
- Best case isn't usually helpful
  - This is because the best case is almost always $O(1)$
- Average case can be helpful (typically requires probabilistic analysis to "prove" it)
- Worst case is the most important, usually

Learning Complexity Analysis

- Analyzing a complex algorithm is hard; more in CS 483
  - Most analyses in here will be straight-forward
  - Mostly use the common patterns that were given above
- If you haven't got a clue looking at the code, trying running it and checking

Take-Home

- Today: order analysis captures the big picture of algorithm complexity
  - Different functions grow at different rates
  - Big O: upper bound (there are also other bounds)
- Next time
  - Reading: finish Chapter 5, Chapter 15
  - Practice: Exercises 5.39 and 5.44 which explore string concatenation