Objectives

In this chapter, we'll consider:

- Computational and Asymptotic Complexity
- Big-O Notation
- Properties of the Big-O Notation
- Examples of Complexities
- Finding Asymptotic Complexity

1

Computational and Asymptotic Complexity

- ***Algorithms*** are an essential aspect of data structures
- Data structures are implemented using algorithms
- Some algorithms are more efficient than others
- An algorithm's ***complexity*** is a function describing the efficiency of the algorithm in terms of the amount of data the algorithm must process
- There are two main complexity measures of efficiency
  - Time Complexity
  - Space Complexity

2

Computational and Asymptotic Complexity (continued)

- ***Time complexity*** describes the amount of time an algorithm takes in terms of the amount of input
- ***Space complexity*** describes the amount of memory (space) an algorithm takes in terms of the amount of input
- For both measures, we are interested in the algorithm's ***asymptotic*** complexity. This asks: when *N* (number of input items) goes to infinity, what happens to the algorithm's performance?

3

3

Computational and Asymptotic Complexity (continued)

- Consider the function $f(n) = n^2 + 100n + \log_{10}n + 1000$
- As the value of *n* increases, the importance of each term shifts until for large *n*, only the n^2 term is significant

n	f(n)	n^2		100n		$\log_{10}n$		1,000	
	Value	Value	%	Value	%	Value	%	Value	%
1	1,101	1	0.1	100	9.1	0	0.0	1,000	90.83
10	2,101	100	4.76	1,000	47.6	1	0.05	1,000	47.60
100	21,002	10,000	47.6	10,000	47.6	2	0.001	1,000	4.76
1,000	1,101,003	1,000,000	90.8	100,000	9.1	3	0.0003	1,000	0.09
10,000	101,001,004	100,000,000	99.0	1,000,000	0.99	4	0.0	1,000	0.001
100,000	10,010,001,005	10,000,000,000	99.9	10,000,000	0.099	5	0.0	1,000	0.00

4

4

Big-O Notation

- The most commonly used notation for asymptotic complexity used is "big-O" notation
- In the previous example we would say:
  $n^2 + 100n + \log_{10}n + 1000 = O(n^2)$ (read "big-O of n squared")

Definition: Let $f(n)$ and $g(n)$ be functions, where $n \in Z^+$ is a positive integer.

We write $f(n) = O(g(n))$ if and only if:
There exists some fixed real number c and positive integer N satisfying:

$$0 \leq f(n) \leq cg(n) \text{ for all } n \geq N.$$

(And we say, "f of n is big-oh of g of n.")

- This means that functions like $n^2 + n$, $4n^2 - n \log n + 12$, $n^2/5 - 100n$, and so forth are all $O(n^2)$ because there's a value of c (say, c = 5) where all of these expressions are less than $5 * n^2$ for all n > 2.

5

5

Big-O Notation

- But what if the calculated time an algorithm takes is more complicated? Consider the case we had before:
- function $f(n) = n^2 + 100n + \log_{10}n + 1000$
- As we saw, when n gets very larger and larger, the biggest term in the expression (in this case, n^2) dominates the result, making the other, smaller terms, shrinking in importance to virtually nil
- So we call the algorithm with this computational complexity an $O(n^2)$ algorithm
- So, to sum up, given the f(n) expression, ignore all but the dominant term as n get larger and larger, AND also discard any constant coefficient on that term.
- Example: $f(n) = 7n^3 + 100n^2 + 1000n$ becomes just $O(n^3)$

6

6

Big-O Notation

- A big-O example can be seen in Linear Search, an algorithm for finding a value in a list or array.
- Every element in the list or array is inspected until one of two things happens:
  - The desired value is found
  - The end of the list or array is reached
- We assume that examining each element takes a constant amount of time, which we'll call k
- If the list or array has n elements in it, then in the best case we'll find the desired value in the first element and be done!
- In the best case, we find the element in the first position, which takes k time to perform
- In the worst case, we have to visit every element in the list or array, which takes k * n time to perform
- So in the best case, it's "constant" time (f(n) = k), whereas in the worst case, it's "linear time" (f(n) = k * n)

7

7

Big-O Notation

- In the best case, "constant" time (f(n) = k)
- A constant k is the same as k * n^0, where n is the size of the array
- Big-O notation does not concern itself with the constant time on the polynomial term, so it's O(n^0)
- However, NOBODY writes it that way – it's instead written as O(1)
- All constant-time algorithms are described as O(1) algorithms
- In the worst case, Linear Search is "linear" time (f(n) = k * n) if every element in the list or array is visited during the search
- Again, Big-O notation does not concern itself with the constant time on the polynomial term, so it's O(n^1) or just O(n)

8

8

Big-O Notation (continued)

- More formally, we present this example:
- We want a constant value c and a large enough N such that:

$$3n^2 + 4n - 2 \leq cn^2$$

for all $n \geq N$

- Dividing by n^2 gives us

$$3 + 4/n - 2/n^2 \leq c$$

- Choosing $N = 1$, we need to find a c such that

$$3 + 4 - 2 \leq c$$

- We can set $c = 6$, so we have

$$3n^2 + 4n - 2 \leq 6n^2$$

for all $n \geq 1$

- So our function is $O(n^2)$ because c = 6 works for all n
- Big-O provides a formal method for expressing ***asymptotic upper bounds***, bounding the growth of a function from above

9

9

Big-O Notation (continued)

- Knowing where a function lies within the big-O hierarchy lets us compare it quickly with other functions, while not concerning ourselves with the particulars of processor speed and the specific constant number of operations occurring for each value from 1 to n
- Thus we can get a general idea of which algorithm has the best time performance. If two algorithms have the same big-O complexity, we can then get more specific
- But if different algorithms have different big-O complexities, we can quickly identify which one will have the better performance as the size n of the input approaches gigantic values

10

10

Properties of Big-O Notation

- The following is a list of useful theorems you can use to simplify big-O calculations
- Big-O is *transitive*:
  if $f(n) = O(g(n))$ and $g(n)$ is $O(h(n))$, then $f(n) = O(h(n))$
- If $f(n) = O(h(n))$ and $g(n)$ is $O(h(n))$, then $f(n) + g(n) = O(h(n))$
- Any k^{th} degree polynomial of n is also $O(n^{k+j})$ for any $j > 0$

11

11

Properties of Big-O Notation (continued)

- $f(n) = O(g(n))$ is true if $\lim_{n\to\infty} f(n)/g(n)$ is a constant.
  Put another way, if $f(n) = cg(n)$ for some positive value of c, then $f(n) = O(g(n))$
- $\log_a n = O(\log_b n)$ for any log bases $a, b > 1$. This means, except for a few unusual cases, we don't care what base our logarithms are. Why? Because you can convert between different log bases by simply multiplying by a constant!
- Given the preceding, we can use just any base and rewrite the relationship as $\log_a N = O(\lg N)$ for positive $a > 1$
  (where "lg n" is a common shorthand for $\log_2 n$
- This means that O(log N) and O(lg N) are identical

12

12