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

1

Big-O Notation – Linear Search

- Suppose that examining each element takes a constant amount of time, which we can call k (where k is a constant)
- If the list of array has n elements in it, then in the best case we'll find the desired value in the first element and be done! This takes k time to perform, independent of n.
- In the worst case, we have to visit every element in the list or array, which takes a total of k * n time to perform
- The worst case is the important one here (as it usually is in most algorithms), so we say that Linear Search takes O(n) time
- This meets the definition of big-O, because we can simply choose a value of c that is greater than k, so we get:
  $T(n) = k * n < c * n$ for values of n greater than some N

2

2

Big-O Notation – Binary Search

- Binary Search of a sorted list is more efficient than Linear Search of an unsorted list
- We assume each step takes constant time k, and we can discard half of the array's contents and continue our search in the other half
- In the worst case, we only have to visit enough elements in the array, cutting the search space in half each time, to reduce the search space down to just one element – that's lg n, so the total time it takes is k * lg n
- This is a logarithmic relationship between size of the array and the number of elements to be visited, so we say that Binary Search takes O(lg n) or O(log n) time
- This meets the definition of big-O – there's a c > k where:
  T(n) < c * log n for values of n greater than some N

3

3

Big-O Notation

- 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, then we can then try to get more specific to determine which algorithm is more efficient
- 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 becomes very large

4

4

Big-O Notation – Recurrence Relations

- Linear Search takes O(n) time
- So how is the time it takes to perform a Linear Search affected by the value of n, in other words, when the list or array gets larger?
- We can intuitively understand that if a list should become, say, twice as long, then in the worst case a Linear Search will also take twice as long
- If we use T(n) to represent the time to perform the worst case Linear Search, then we have the following relation:
  $T(n) = 2 * T(n / 2)$
- Also, if the list or array becomes exactly one element longer, then the relation there would be:
  $T(n) = T(n – 1) + k$ (k is the time to look at one element)

5

5

Big-O Notation – Recurrence Relations

- Relations like $T(n) = 2 * T(n/2)$ and $T(n) = T(n–1) + k$ are called recurrence relations
- Both recurrence relations above have the same solution:
  $T(n) = k * n$ for some constant k, and is therefore O(n)
- This solution implies an initial condition of $T(0) = 0$, which we can assume is true – after all, if the array's size is zero, then there's nothing to search and that takes zero time ☺
- Another initial condition is that $T(1) = k$
- Recurrence relations can be tricky to solve mathematically! The math involved in solving general recurrence relations is beyond the scope of this class.
- So we will only deal with a few basic cases of them in this class. In CS 312, they will be discussed in greater detail.

6

6

Big-O Notation – Recurrence Relations

- Common Recurrence Relations that we will use -- these are informal definitions, and not mathematically rigorous! But they're useful in recognizing where an algorithm fits in the general categories of big-O notation we'll be examining.

- T(n) = T(n / 2) + k (we'll use k to represent O(1) time)
  When the size of the structure doubles, the time goes up only by some constant value.
  This corresponds to T(n) is O(log n), or logarithmic time

- T(n) = T(n – 1) + k or
  T(n) = T(n / 2) * 2 + k
  When the size of the structure is incremented, the time goes up by a constant amount.
  When the size of the structure doubles, the time also doubles (with or without a constant increase as well).
  This corresponds to T(n) is O(n), or linear time

7

7

Big-O Notation – Recurrence Relations

- T(n) = 2 * T(n / 2) + O(n)
  When the size of the structure doubles, the time doubles and also adds a linear component
  This corresponds to T(n) is O(n log n), or linearithmic time

- T(n) = T(n – 1) + O(n)
  When the size of the structure is incremented, the time increases by a linear value
  This corresponds to T(n) is $O(n^2)$, or quadratic time

- T(n) = T(n – 1) * k
  When the size of the structure is incremented, the time is multiplied by a constant value
  This corresponds to T(n) is $O(k^n)$, or exponential time

8

8

Properties of Big-O Notation

- Review of properties of Big-O Notation
- Big-O is *transitive*:
  if $f(n)$ is $O(g(n))$ and $g(n)$ is $O(h(n))$, then $f(n)$ is $O(h(n))$
- If $f(n)$ is $O(h(n))$ and $g(n)$ is $O(h(n))$, then $f(n) + g(n)$ is $O(h(n))$
- A function an^k is $O(n^k)$ for any constant $a > 0$
- Any k^{th} degree polynomial is also $O(n^{k+j})$ for any $j > 0$

9

9

Ω and Θ Notations

- Big-O only gives us the upper bound of a function
- If we ignore constant factors and let n get big enough, Big-O means a given function will never become bigger than some other function
- This can give us too much freedom – example: a sorting algorithm that is $O(n^2)$ is also technically $O(n^3)$, because n^2 itself is also $O(n^3)$ - however, $O(n^2)$ is the more meaningful upper bound (the least upper bound) that we care about
- We could also define a ***lower bound***, a function that always grows more slowly than $T(n)$, and a ***tight bound***, a function that grows at about the same rate as $T(n)$

10

10

Ω and Θ Notations

- Big-Ω is for lower bounds just as big-O is for upper bounds
- **Definition**: Let $f(n)$ and $g(n)$ be functions
  We write $f(n) = \Omega(g(n))$ if and only if $g(n) = O(f(n))$.
  We say "f of n is omega of g of n."
- So g is a lower bound for f ; after a certain N, and without regard to multiplicative constants, f will never go below g

- Finally, theta (or Θ) notation combines upper bounds with lower bounds to get a tight bound

  Definition: Let $f(n)$ and $g(n)$ be functions, where n is a positive integer. We write $f(n) = \Theta(g(n))$ if and only if $f(n) = O(g(n))$ and $g(n) = O(f(n))$. Thus f and g are "tightly bound" to each other. We say "f of n is theta of g of n."

11

11

Logarithmic, Polynomial, and Exponential Complexity

- An algorithm that runs in polynomial time $O(n^k)$ is (for large enough n) preferable to an algorithm that runs in exponential time $O(a^n)$ for some constant a
- So an algorithm that runs in polynomial time is (for large enough n) preferable to an algorithm that runs in exponential time

- A similar relationship between logarithmic and polynomial time: $(\log n)^\varepsilon$ is better than $O(n^k)$ for any positive constants k and ε, for sufficiently large value of n
- So an algorithm that runs in logarithmic time is (for large enough n) preferable to an algorithm that runs in polynomial time

12

12

Logarithmic, Polynomial, and Exponential Complexity

- Since we examine algorithms in terms of their time (and space) complexity, we can classify them this way, too

Class	Complexity	Number of Operations and Execution Time (1 instr/μsec)					
n		10		10^2		10^3	
constant	$O(1)$	1	1 μsec	1	1 μsec	1	1 μsec
logarithmic	$O(\lg n)$	3.32	3 μsec	6.64	7 μsec	9.97	10 μsec
linear	$O(n)$	10	10 μsec	10^2	100 μsec	10^3	1 msec
$O(n \lg n)$	$O(n \lg n)$	33.2	33 μsec	664	664 μsec	9970	10 msec
quadratic	$O(n^2)$	10^2	100 μsec	10^4	10 msec	10^6	1 sec
cubic	$O(n^3)$	10^3	1 msec	10^6	1 sec	10^9	16.7 min
exponential	$O(2^n)$	1024	10 msec	10^{30}	$3.17 * 10^{17}$ yrs	10^{301}	