DATA STRUCTURES and ALGORITHMS in C++

FOURTH EDITION

Chapter 2: Complexity Analysis

Objectives

Looking ahead – in this chapter, we'll consider:

- Computational and Asymptotic Complexity
- Big-O Notation
- Properties of the Big-O Notation
- Ω and Θ Notations
- Possible Problems with the Notation

Objectives (continued)

- Examples of Complexities
- Finding Asymptotic Complexity
- Best, Average, and Worst Cases
- Amortized Complexity
- NP-Completeness

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
- Efficiency is preferred; we need metrics to compare them
- 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

Computational and Asymptotic Complexity

- ***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?

Computational and Asymptotic Complexity

- To illustrate this, consider $f(n) = n^2 + 100n + \log_{10} n + 1000$
- 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

Fig. 2-1 The growth rate of all terms of function $f(n) = n^2 + 100n + \log_{10} n + 1{,}000$.

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-oh of n squared")

Definition: Let $f(n)$ and $g(n)$ be functions, where $n \varepsilon Z$ is a positive integer. We write $f(n) = O(g(n))$ if and only if there exists a real number c and positive integer N satisfying $0 \leq f(n) \leq cg(n)$ for all $n \geq N$. (And we say, "f of n is big-oh of g of n.“)

- This means that functions like $n^2 + n$, $4n^2 - n \log n + 12$, $n^2/5 - 100n$, $n \log n$, and so forth are all $O(n^2)$

Properties of Big-O Notation

- The following is a list of useful facts 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 $f(n) + g(n) = O(h(n))$
  - A function $an^k = O(n^k)$ for any $a > 0$
  - Any k^{th} degree polynomial is $O(n^{k+j})$ for any $j > 0$

Properties of Big-O Notation (continued)

- $f(n) = O(g(n))$ is true if $\lim_{n->\infty} f(n)/g(n)$ is a constant. Put another way, if $f(n) = cg(n)$, then $f(n) = O(g(n))$
- $\log_a n = O(\log_b n)$ for any $a, b > 1$. This means, except for a few cases, we don't care what base our logarithms are
- Given the preceding, we can use just one base and rewrite the relationship as $\log_a n = O(\lg n)$ for positive $a \neq 1$ and $\lg n = \log_2 n$

Ω and Θ Notations

- Big-O only gives us the upper bound of a function
- So if we ignore constant factors and let n get big enough, some function will never be bigger than some other function
- This can give us too much freedom
- Consider that selection sort is $O(n^3)$, since n^2 is $O(n^3)$ - but $O(n^2)$ is a more meaningful upper bound
- We need a ***lower bound***, a function that always grows more slowly than $f(n)$, and a ***tight bound***, a function that grows at about the same rate as $f(n)$
- Section 2.4 gives a good introduction to these concepts; let's look at a different way to approach this

Ω and Θ Notations (continued)

- Big-Ω is for lower bounds what big-O is for upper bounds

  Definition: Let $f(n)$ and $g(n)$ be functions, where n is a positive integer. 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 notation combines upper bounds with lower bounds to get 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 $g(n) = O(f(n))$ and $g(n) = 0(f(n))$. We say "f of n is theta of g of n."

Examples of Complexities

- Since we examine algorithms in terms of their time and space complexity, we can classify them this way, too
- This is illustrated in the next figure

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}	

Fig. 2.4 Classes of algorithms and their execution times on a computer executing 1 million operations per second

(1 sec = 10^6 μsec = 10^3 msec)

Examples of Complexities (continued)

n		10^4		10^5		10^6	
constant	$O(1)$	1	1 µsec	1	1 µsec	1	1 µsec
logarithmic	$O(\lg n)$	13.3	13 µsec	16.6	7 µsec	19.93	20 µsec
linear	$O(n)$	10^4	10 msec	10^5	0.1 sec	10^6	1 sec
$O(n \lg n)$	$O(n \lg n)$	$133 * 10^3$	133 msec	$166 * 10^4$	1.6 sec	$199.3 * 10^5$	20 sec
quadratic	$O(n^2)$	10^8	1.7 min	10^{10}	16.7 min	10^{12}	11.6 days
cubic	$O(n^3)$	10^{12}	11.6 days	10^{15}	31.7 yr	10^{18}	31,709 yr
exponential	$O(2^n)$	10^{3010}		10^{30103}		10^{301030}	

Fig. 2.4 (concluded)

Finding the Complexity

- As we have seen, asymptotic bounds are used to determine the time and space efficiency of algorithms
- Generally, we are interested in time complexity, which is based on assignments and comparisons in a program
- We'll focus on assignments for the time being
- Consider a simple loop:

```
for (i = sum = 0; i < n; i++)
    sum = sum + a[i]
```

- Two assignments are executed once (`sum = 0` and `i = sum`) during initialization
- In the loop, `sum = sum + a[i]` is executed *n* times

Finding Asymptotic Complexity (continued)

- In addition, the `i++` in the loop header is executed *n* times
- So there are 2 + 2*n* assignments in this loop's execution and it is *O*(*n*)
- Typically, as loops are nested, the complexity grows by a factor of *n*, although this isn't always the case
- Consider

```
for (i = 0; i < n; i++) {
   for (j = 1, sum = a[0]; j <= i; j++)
      sum += a[j];
   cout << "sum for subarray 0 through " << i
        <<" is "<<sum<<end1;
}
```

Finding Asymptotic Complexity (continued)

- The outer loop initializes `i`, then executes *n* times
- During each pass through the loop, the variable `i` is updated, and the inner loop and `cout` statement are executed
- The inner loop initializes `j` and `sum` each time, so the number of assignments so far is 1 + 3n
- The inner loop executes *i* times, where *i* ranges from 1 to *n* – 1, based on the outer loop (when *i* is 0, it doesn't run)
- Each time the inner loop executes, it increments `j`, and assigns a value to `sum`
- So the inner loop executes $\sum_{i=1}^{n-1} 2i = 2(1 + 2 + \ldots + n - 1) = 2n(n - 1)$ assignments

Finding Asymptotic Complexity (continued)

- The total number of assignments is then $1 + 3n + 2n(n - 1)$, which is $O(1) + O(n) + O(n^2) = O(n^2)$
- As mentioned earlier, not all loops increase complexity, so care has to be taken to analyze the processing that takes place
- However, additional complexity can be involved if the number of iterations changes during execution
- This can be the case in some of the more powerful searching and sorting algorithms

Best, Average, and Worst Cases

- If we want to truly get a handle on the complexity of more complicated algorithms, we need to distinguish three cases:
  - ***Worst case*** – the algorithm takes the maximum number of steps
  - ***Best case*** – the algorithm takes the fewest number of steps
  - ***Average case*** – performance falls between the extremes
- For simple situations we can determine the average case by adding together the number of steps required for each input and dividing by the number of inputs
- However, this is based on each input occurring with equal probability, which isn't always likely

Best, Average, and Worst Cases (continued)

- To be more precise, we need to weight the number of steps that occur for a given input by the probability of that input occurring, and sum this over the number of inputs:

$$\sum_i p(input_i)\, steps(input_i)$$

- In probability theory, this defines the ***expected value***, which assumes the probabilities can be determined and their distribution known
- Because *p* is a probability distribution, it satisfies two constraints:
  - The function *p* can never be negative
  - The sum of all the probabilities is equal to 1

Best, Average, and Worst Cases (continued)

- Consider the example of sequentially searching an unordered array to find a target value
- The best and worst cases are straightforward:
  - Best case occurs when we find the target in the first cell
  - Worst case occurs when we find the target in the last cell, or not at all (but end up searching the entire array)
- For the average case, we first have to consider the probability of finding the target
- If we assume a uniform distribution of *n* values, then the probability of finding the target in any one location is $\frac{1}{n}$

Best, Average, and Worst Cases (continued)

- So we would find the target in the first location with $p = 1/n$, in the second location with $p = 1/n$, etc.
- Since the number of steps required to get to each location is the same as the location itself, our sum becomes:

$$1/n * (1 + 2 + \ldots + n) = (n + 1) / 2$$

- Again, this is based on an equally likely chance of finding the target in any cell
- If the probabilities differ, then the computation becomes more involved

Selection Sort

Selection Sorting Algorithm:

- During the j-th pass (j = 0, 1, …, n – 2), we will examine the elements of the array a[j] , a[j+1], …, a[n-1] and determine the index min of the smallest key.
- Swap a[min] and a[j].

```
selection_sort(int_array a) {
     if (a.size() == 1) return;
     n = a.size();
     for (int j = 0; j < n - 1; ++j) {
       min = j;
      for (int k= j+1; k<=n-1; ++k)
        if (a[k] < a[min]) min = k;
      swap a[min] and a[j];
   }
  }
```

http://math.hws.edu/TMCM/java/xSortLab/

Analysis of selection sorting

Consider the program to find the min number in an array:

```
min = 0;
for (j = 1; j < n; ++j)
  if (A[j] > min) min = j;
```

The number of comparisons performed is n – 1.

loop starts with j = 1 and ends with j = n so the number of iterations = n – 1.

In each iteration, one comparison is performed.

Selection sorting – analysis

The inner loop:

n – 1 comparisons during the first iteration of the inner loop

n – 2 comparisons during the 2nd iteration of the inner loop

. . . .

1 comparison during the last iteration of the inner loop

Total number of comparisons = 1 + 2 + … + (n – 1) =

n(n – 1)/ 2 (best as well as the worst-case)