<u>Complexity of Algorithms</u> (Rosen, 6th edition, Section 3.3)   Carol Zander

Consider sorting. When the collection of data that we are sorting is large, it is very important that an efficient algorithm is used, since we would otherwise spend more time than necessary sorting. The algorithms discussed can be applied to any type of objects, including integers, floating point numbers, strings, and complex objects. In general, efficient algorithms are more important than coding tricks.

We saw previously that binary search could find an item in a list much faster than sequential search, but only if the list is sorted. Precisely, what is the difference in their complexity? At some point, you will mostly figure complexity intuitively, but in this section, it is done in a more formal manner. Some of the common complexities are those discussed in Section 3.2 as common running times. Table 1 from your text lists common algorithm complexities.

TABLE 1 Commonly Used Terminology for the Complexity of Algorithms.

Complexity	Terminology
$\Theta(1)$	Constant complexity
$\Theta(\log n)$	Logarithmic complexity
$\Theta(n)$	Linear complexity
$\Theta(n \log n)$	$n \log n$ complexity
$\Theta(n^b)$	Polynomial complexity
$\Theta(b^n)$, where $b > 1$	Exponential complexity
$\Theta(n!)$	Factorial complexity

Consider code that does not depend on input (i.e., no loops). All the following kinds of code are tight O(1), constant time. Input is not relevant on the time taken to execute. No matter the size of the input, the time to execute remains the same. Examples include

- Input/output, e.g., `cout << "The answer is " << result << endl;`
- Arithmetic, e.g., `z = x +y;`
- Assignment, or any expression, e.g., `a = b;  a == b;`
- If statement, e.g., `if (some expression) { some constant code, i.e., no loops }`

Let's look at loops now. Consider a straightforward single *for* loop:

<u>Example 1</u> – Simple single loop
```
int sum = 0;
for (int i = 0; i < n; i++ ) { something O(1) }
```

$$T(n) = O\left(1 + \sum_{i=0}^{n-1} 1 \right) = O\left(1 + \sum_{i=1}^{n} 1 \right) = O(1+n) = O(n)$$

The first 1 is for initialization before the loop. It is not essential to include this because of the nature of big-O (it will not appear in the final answer, is absorbed by the constant). The sum is easier to analyze if it starts at 1 instead of zero, and since we are just summing something that is O(1), we can adjust the summation limits.

Example 2 – Straightforward nested loop
```
    for (int i = 0; i < n; i++ ) {
       for (int j = 0; j < n; j++) {
          something O(1)
       }
    }
```

```
         n-1  n-1              n    n              n
         ---  ---              ---  ---            ---
         \    \                \    \              \
T(n) = O( /    /   1 ) = O(  /    /   1 ) = O(  /   n = O(n+n+n+...+n) = O(n²)
         ---  ---              ---  ---            ---
         i=0  j=0              i=1  j=1            i=1
```

Example 3 – What if inner loop stops at outer loop's variable?
```
    for (int i = 0; i < n; i++ ) {
       for (int j = 0; j < i; j++) {
          something O(1)
       }
    }
```

```
         n-1   i               n   i-1             n                n     n
         ---  ---              ---  ---            ---              ---   ---
         \    \                \    \              \                \     \
T(n) = O( /    /   1 ) = O( /    /   1 ) = O( /  (i-1) = O( /  i  - /   1 )
         ---  ---              ---  ---            ---              ---   ---
         i=0  j=0              i=1  j=1            i=1              i=1   i=1
```

```
     = O(n(n+1)/2 - n) = O(n²)
```

Example 4 – What if the inner loop starts at outer loop's variable?
```
    for (int i = 0; i < n; i++ ) {
       for (int j = i; j < n; j++) {
          something O(1)
       }
    }
```

```
         n-1    n               n    n              n                n              n
         ---  ---              ---  ---            ---              ---            ---
         \    \                \    \              \                \              \
T(n) = O( /    /   1 ) = O(  /    /   1 ) = O( /  (n-i+1)) = /  (n+1)  - /   i
         ---  ---              ---  ---            ---              ---            ---
         i=0  j=i              i=1  j=i            i=1              i=1            i=1
```

```
     = O((n+1)n - n(n+1)/2) = O(n²)
```

Consider the highlighted summation that doesn't start at one, the j summation. This is the tail end of the summation from 1 to n since i is somewhere between 1 and n. Here's how to handle it.

Consider the whole sum from 1 to n, n terms in total. Then consider the sum from 1 to i-1, the first (i-1) terms. If we subtract these summations, what's left are the terms at the end of the summation, the terms from i to n, which are the ones we want (numbers represent the terms being added):

```
  1 + 2 + 3 + 4 + ... + i-1  +   i + i+1 + ... + n-1 + n
  _____/   _____/
     the sum from 1 to i-1           the sum of what we want
  _____/
               the sum from 1 to n
```

$$= \sum_{i=1}^{n} \left(\sum_{j=1}^{n} 1 - \sum_{j=1}^{i-1} 1 \right) \qquad \text{Now we can use our known formula.}$$

$$= \sum_{i=1}^{n} (n - (i - 1)) = \sum_{i=1}^{n} (n - i + 1)$$

You might need to do some reasoning along with the summations. For example, suppose you have a nested loop and the inner loop is inside an if that tests for an even number (e.g., `if (i % 2 == 0)`). The inner loop will only be executed half as many times. You can describe this situation (in text) and then show the inner summation multiplied by 1/2 .

<u>Example 5</u> – What about something like the binary search?

```
while (low <= high) {
   middle = (low+high)/2;
   if (target == a[middle])
      // save middle and return;
   else
      // test for < and > and adjust high or low appropriately
}
```

Assume without loss of generality that $n = 2^k$ where k is the power of 2 that makes 2^k larger than or equal to the actual number of items to search (legal to assume a larger number of items). For example, if $n = 100$, let $k = 7$ since $2^7 = 128 > 100$. In worse case, when the item is not in the array, the loop is executed k times. For example, consider $n = 2^7$. The array elements to consider each time through the loop are
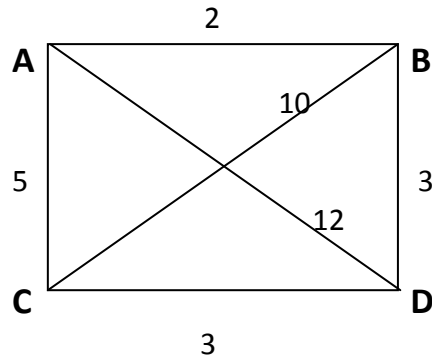
$$2^7$$
$$2^7/2 = 2^6$$
$$2^6/2 = 2^5$$
$$2^5/2 = 2^4$$
$$2^4/2 = 2^3$$
$$2^3/2 = 2^2$$
$$2^2/2 = 2$$
$$2/2 = 1$$

The loop is exited after 7 , or in general, k, times. Since $n = 2^k$, $k = \log_2 n$, so the complexity is $O(\log_2 n)$.

<u>Example 6</u> – The Traveling Salesperson problem

Consider one more type of problem, exponential.

The problem: Given cities, find the shortest route in which salesperson visits each city one time, starting and ending in the same city. For example

Choice of cities is $\underline{4} \cdot \underline{3} \cdot \underline{2} \cdot \underline{1} = 24$ After choosing the first city to visit, there are 3 choices left. After choosing the next city, there are 2 cities left to choose from. Then only one is left. There are 24 possible combinations:

```
a  b  c  d
a  b  d  c
a  c  b  d
a  c  d  b
a  d  b  c
a  d  c  b
b

↓

b


c

↓

c


d

↓

d

```

The complexity is $O(n!) = O(n^n)$. Problems of this complexity are called intractable, NP-hard (non-deterministically polynomial, meaning given a solution it can be checked for correctness in polynomial time (in a non-deterministic manner)).

Using dynamic programming, a technique that breaks down a problem into smaller pieces recursively, and using exponential space, it has been shown to be $O(n^2 2^n)$.

Often exponential growth problems are solved using brute force, similar to this problem, by trying all possibilities.