VI. Big O

Big O - metric we use to describe the efficiency of algorithms

Time Complexity

: ~(asymptotic runtime, or big O time)

Big O, Big Theta, and Big Omega

- **O (Big O)** - upper bound on the time
    - e.g. an algorithm that prints all the values in an array - O(N), O(N^2)... etc
- **Ω (Big Omega)** - lower bound
    - e.g. an algorithm that prints all the values in an array - Ω(N), Ω(log N)... etc
- **θ (Big Theta)** - both O and Ω - tight bound
    - " - θ(N)

Tip

- in industry, big O is closer to what academics mean by θ.
- try to offer the tightest description of the runtime

Best Case, Worst Case, and Expected Case

: Let's think of a quick sort - picks a random element as a "pivot" and swaps values in the array such that the elements less than pivot appear before elements greater than pivot

- **Best Case** - if all elements are equal, then quick sort will just traverse through the array once - O(N)
- **Worst Case** - if the pivot is repeatedly the biggest element in the array - O(N^2)
- **Expected Case** - O(N log N)

Tip

- Best case is rarely discussed - not very useful concept
- In most cases, the worst case and the expected case are the same
- No particular relationship between best/worst/expected case <-> big O/theta/omega

Space Complexity

: Also need to take memory or space into account

e.g. an array of size n -- O(n), a two-dimensional array of size nxn -- O(n^2)

- Stack, recursive calls --> O(n) time & O(n) space

```
int sum(int n) {
    if (n <= 0) {
        return 0;
    }
    return n + sum(n-1);
}
```

  - e.g. sum(4)

```
sum(4)
    -> sum(3)
        -> sum(2)
            -> sum(1)
                -> sum(0)

--> each of these calls is added to the call stack, taking up actual memory
```

- Adding adjcanet elements between 0 and n

```
int pairSumSequence(int n) {
    int sum = 0;
    for (int i = 0; i < n; i++) {
        sum += pairSum(i, i+1);
    }
    return sum;
}

```

```
int pairSum(int a, int b) {
    return a + b;
}

// will be roughly O(n) calls to pairSum but these calls don't exist simultaneously
// need O(1) space
```

Drop the Constants

: Big O just describes the rate of increase --> doesn't mean O(N) is always better than O(N^2)

- O(2N) ~ O(N)

```
int min = Integer.MAX_VALUE;
int max = Integer.MIN_VALUE;
for (int x : array) {
    if (x < min) min = x;
    if (x > max) max = x;
}
```

```
int min = Integer.MAX_VALUE;
int max = Integer.MIN_VALUE;
for (int x: array) {
    if (x < min) min = x;
}
for (int x: array) {
    if (x > max) max = x;
}
```

Drop the Non-Dominant Terms

- O(N^2 + N) ~ O(N^2)
- O(N + log N) ~ O(N)

- O(5 x 2N + 1000N^100) ~ O(2^N)

However, for the ones with multiple variables, expression cannot be reduced without detailed knowledge of the variables

- e.g. O(B^2 + A)

```
for (int a : arrA) {
    print(a);
}

for (int b : arrB) {
    print(b);
}

// O(A + B) -- do 'A' works then B
```

```
for (int a : arrA) {
    for (int b : arrB) {
        print(a + ",", + b);
    }
}

// O(A * B) -- do 'B' work for each element in A
```

Amortized Time

- Consider an ArrayList (dynamically resizing array)
    - When the array hits capacity, the ArrayList class will create a new array with double the capacity and copy all the elements over to the new array
    - As we insert elements, we double the capacity when the size of the array is a power of 2
        - array sizes: 1, 2, 4, 8, 16, ..., X
            - Adding right to left --> **x + x/2 + x/4 + ... + 1 ~ 2X**

    - X insertion takes O(2X) time --> **Amortized Time for each insertion is O(1)**

Log N Runtimes

- Eg. Binary search in an N-element sorted array

```
search 9 within {1, 5, 8, 9, 11, 13, 15, 19, 21}
   compare 9 to 11 --> smaller
   search 9 within {1, 5, 8, 9, 11}
     compare 9 to 8 --> bigger
     search 9 within {9, 11}
       compare 9 to 9
       return
```

- Total runtime is a matter of how many steps (dividing N by 2 each time) we can take until N becomes 1
- Suppose we have 16 elements
  - N = 16 --> N = 8 --> N = 4 --> N = 2 --> N = 1 ==> 4 steps down to 1
  - O (log 16) runtime

Recursive Runtimes

```
int f (int n) {
    if (n <= 1) {
        return 1;
    }
    return f(n - 1) + f (n - 1);
}
```

- Consider we have n = 4
  - Calls f(4) --> 2^0
    - f(4) calls f(3) twice --> 2^1
      - each f(3) calls f(2) twice --> 2^2
        - each f(2) calls f(1) then return --> 2^3

--> We have 2^0 + 2^1 + 2^2 + ... + 2^N = 2^(N+1) nodes

- However, the **space complexity** of this algorithm will be O(N) since O(N) exist at any given time