Lecture 3: Efficiency, Analysis, and Order Part 2

CS3310

In-Class Exercise

1. What is the time complexity T(n) of the nested loops below? Assume n is a power of 2

```
for (i = 1; i <= n; i++)
     j = n;
     while ( j > 1)
          <body of while loop> // takes constant time (1)
          j = ⌊j / 2⌋
```

2. What is the time complexity T(n) of the following algorithm? Assume n is divisible by 4.

```
for (i = 2; i <= n; i++)
     for (j = 0; j <= n)
          cout << i << j;
          j = j + ⌊n / 4⌋;
```

Time Complexity Example

What is the time complexity of the following algorithm, assuming *n* is divisible by 2?

```
for (i = 1; i <= 1.5n; i++)
    cout << i;
for (i = n; i >= 1; i--)
    cout << i;
```

What is the basic operation?

Time Complexity Example

What is the time complexity of the following algorithm, assuming *n* is divisible by 2?

```
for (i = 1; i <= 1.5n; i++)
    cout << i;
for (i = n; i >= 1; i--)
    cout << i;
```

What is the basic operation? There are **two** basic operations, the printing of *i* in each loop

Does this algorithm have an every-case time complexity?

Time Complexity Example

What is the time complexity of the following algorithm, assuming n is divisible by 2?

```
for (i = 1; i <= 1.5n; i++)
    cout << i;
for (i = n; i >= 1; i--)
    cout << i;
```

What is the basic operation? There are **two** basic operations, the printing of i in each loop

Does this algorithm have an every-case time complexity? Yes! There is no way to break out of either loop.

What is T(n)?

Time Complexity Example

What is the time complexity of the following algorithm, assuming n is divisible by 2?

```
for (i = 1; i <= 1.5n; i++)
    cout << i;
for (i = n; i >= 1; i--)
    cout << i;
```

What is the basic operation? There are **two** basic operations, the printing of i in each loop

Does this algorithm have an every-case time complexity? Yes! There is no way to break out of either loop.

What is T(n)? 1.5n (first loop) + n (second loop)

T(n) = 2.5n

Insertion Sort Time Complexity

```
for (index i = 2; i <= n; i++)
     target = A[i]
     j = i - 1;
     while (j > 0 and target < A[j])
          A[j + 1] = A[j];
          j--;
     A[j + 1] = target;
```

What is the basic operation?

Insertion Sort Time Complexity

```
for (index i = 2; i <= n; i++)
     target = A[i]
     j = i - 1;
     while (j > 0 and target < A[j])
          A[j + 1] = A[j];
          j--;
     A[j + 1] = target;
```

What is the basic operation? The comparison of `target` and A[j]

Does this algorithm have an every-case time complexity?

Insertion Sort Time Complexity

```
for (index i = 2; i <= n; i++)
     target = A[i]
     j = i - 1;
     while (j > 0 and target < A[j])
          A[j + 1] = A[j];
          j--;
     A[j + 1] = target;
```

What is the basic operation? The comparison of `target` and A[j]

Does this algorithm have an every-case time complexity? **No**! Since we can break out of the `while` loop early, we need to find B(n), A(n), and W(n)

Insertion Sort Time Complexity

```
for (index i = 2; i <= n; i++)
     target = A[i]
     j = i - 1;
     while (j > 0 and target < A[j])
          A[j + 1] = A[j];
          j--;
     A[j + 1] = target;
```

What is the basic operation? The comparison of `target` and `A[j]`

What is the best case scenario?

Insertion Sort Time Complexity

```
for (index i = 2; i <= n; i++)
     target = A[i]
     j = i - 1;
     while (j > 0 and target < A[j])
          A[j + 1] = A[j];
          j--;
     A[j + 1] = target;
```

What is the basic operation? The comparison of `target` and `A[j]`

What is the best case scenario? The array is already sorted:

- The `while` loop immediately exits each time it is reached (`target` will always be less than `A[j]`).
- The `for` loop iterates from 2 to n, so $B(n) = n - 1$

Insertion Sort Time Complexity

```
for (index i = 2; i <= n; i++)
     target = A[i]
     j = i - 1;
     while (j > 0 and target < A[j])
          A[j + 1] = A[j];
          j--;
     A[j + 1] = target;
```

What is the basic operation? The comparison of `target` and `A[j]`

What is the worst case scenario?

Insertion Sort Time Complexity

```
for (index i = 2; i <= n; i++)
     target = A[i]
     j = i - 1;
     while (j > 0 and target < A[j])
          A[j + 1] = A[j];
          j--;
     A[j + 1] = target;
```

What is the basic operation? The comparison of `target` and `A[j]`

What is the worst case scenario? The array is sorted in reverse order:

- The `while` loop iterates from j to 1 n times.
  - In the first iteration of the `for` loop, j starts at 1. In the second iteration, it starts at 2, etc..
- i.e. $1 + 2 + \ldots + n \longrightarrow W(n) = n(n + 1) / 2$

Order

There are an *infinite* number of time complexities, making algorithms difficult to compare.

With **Order**, we group algorithms with other algorithms of *similar* complexity.

- Algorithms with time complexities of n, $100n$, etc. are **linear-time**
  - Their time complexity grows linearly with n
- Algorithms with time complexities of n^2, $0.01n^2$, etc. are **quadratic-time**
  - Their time complexity grows quadratically with n

Every quadratic algorithm is worse than every linear algorithm given a large enough input!

Although n^2 and $0.01n^2$ are very different time complexities, we group them together because they both *grow* in a similar manner (quadratically).

Order

- $5n^2$ and $5n^2 + 100$ are *pure quadratic* functions as they do not contain a linear term
- $0.1n^2 + n + 100$ is a *complete quadratic* function since it contains a linear term, n
  - **Note**: even though we multiply n^2 by 0.1, this term will still eventually dominate since quadratic terms grow much faster than linear ones:

n	$0.1n^2$	$0.1n^2 + n + 100$
10	10	120
20	40	160
50	250	400
100	1,000	1,200
1,000	100,000	101,100

For smaller inputs (10, 20, 50...) $0.1n^2 + n + 100$ is much less efficient than $0.1n^2$.

However, once the input size reaches 100 and above, the impact n and 100 have on performance are negligible.

Complexity Classes

The term that eventually dominates is the one we are interested in.

- In any function, we can throw away lower-order terms:
  - i.e. $0.1n^3 + 10n^2 + 5n + 25$ is a *complete cubic* function. We throw away $10n^2$, $5n$, and 25, as they are each lower-order than $0.1n^3$. As n grows, $0.1n^3$ will eventually dominate the others.
- When a function is cubic, we say that it is in the complexity class $\Theta(n^3)$
  - We also say the function is "order n^3"

Complexity Classes

Common complexity classes, from most efficient to least efficient

$\Theta(1)$ $\Theta(\lg n)$ $\Theta(n)$ $\Theta(n \lg n)$ $\Theta(n^2)$ $\Theta(n^3)$ $\Theta(2^n)$ $\Theta(n!)$

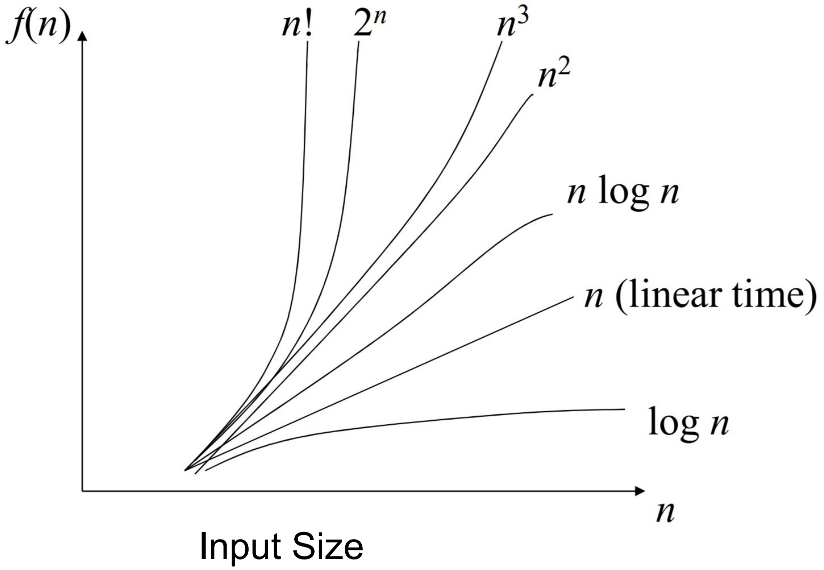
If we know a function's complexity class, we know that it is more efficient than those to its right in this list and less efficient than those to its left.

Example:

- As n grows, any function that is $\Theta(n \lg n)$ is more efficient than any function that is $\Theta(n^2)$ and less efficient than any function that is $\Theta(n)$, etc.

Complexity Classes

This chart displays that rate at which functions within common complexity classes *grow* based on input size



Complexity Class Example

What complexity class does the following function belong in?

$$f(x) = n + n^2 + 2^n + n^4$$

Complexity Class Example

What complexity class does the following function belong in?

$$f(x) = n + n^2 + 2^n + n^4$$

- $\Theta(2^n)$, because 2^n eventually dominates the other terms.
    - We can throw out n, n^2, and n^4.

- While we will usually use this type of intuition to determine a function's complexity class, there is a more rigorous way to prove it.

Big O

- For a complexity function $f(n)$, $O(f(n))$ is the set of all complexity functions $g(n)$ for which there exists some positive real constant c and some nonnegative integer N such that for all $n \geq N$: $g(n) \leq c * f(n)$

- i.e.: for the complexity function n^2, we can prove that another complexity function $g(n)$ is in the set $O(n^2)$ *if* we can find a function we know to be in the set $\Theta(n^2)$ that has the same performance or worse over time (the bigger n gets).
    - c is a constant, so multiplying it with a complexity function that is n^2 results in another function that is n^2
    - If we find **any** linear function that is always as good as or worse than $g(n)$ over time, then we say $g(n) \in O(n)$. We can also say that $g(n)$ is "big O of n"

Big O

Let’s determine the big O of $n^2 + 10n$

- We intuitively know it’s $O(n^2)$, but we can prove it
- We need to find a function that is $\Theta(n^2)$ and pick values for c and N. When this function is multiplied by c, it must always bigger than $n^2 + 10n$ once n reaches size N.
- What’s a function we know for certain is $\Theta(n^2)$?

Big O

Let's determine the big O of $n^2 + 10n$

- We intuitively know it's $O(n^2)$, but we can prove it
- We need to find a function that is $\Theta(n^2)$ and pick values for c and N. When this function is multiplied by c, it must always bigger than $n^2 + 10n$ once n reaches size N.
- What's a function we know for certain is $\Theta(n^2)$?
  - n^2 (for obvious reasons)!

Therefore, we have $n^2 + 10n \leq cn^2$ when $n \geq N$

- To prove $n^2 + 10n$ is $O(n^2)$, we need to determine values for c and N.

An easy way to do this:

- For what value of n does n^2 start to dominate $10n$ (so we can ignore that term)?

Big O

Let's determine the big O of $n^2 + 10n$

- We intuitively know it's $O(n^2)$, but we can prove it
- We need to find a function that is $\Theta(n^2)$ and pick values for c and N. When this function is multiplied by c, it must always bigger than $n^2 + 10n$ once n reaches size N.
- What's a function we know for certain is $\Theta(n^2)$?
  - n^2 (for obvious reasons)!

Therefore, we have $n^2 + 10n \leq cn^2$ when $n \geq N$

- To prove $n^2 + 10n$ is $O(n^2)$, we need to determine values for c and N.

An easy way to do this: For what value of n does n^2 start to dominate $10n$? $\boldsymbol{n = 10}$

$10^2 + 10 \times 10 \leq c10^2$

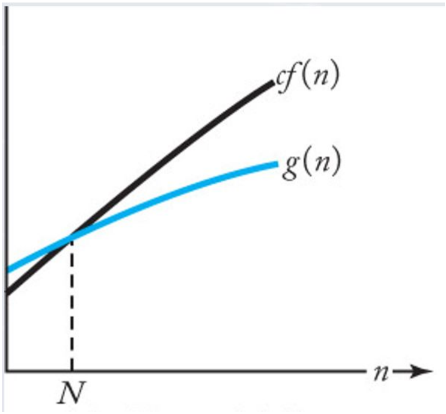
$200 \leq 100c$

$\mathbf{2} \leq c$, so we can pick $c = 2$

Big O

Picking $c = 2$ and N = 10 proves $n^2 + 10n \in O(n^2)$

- i.e. $2n^2$ will always be worse than $n^2 + 10n$ once $n \geq 10$.
- This is our goal in determining *Big O*.



Notice that $g(n)$ can initially be larger than $cf(n)$, but once $n \geq$ N, $g(n)$ is always less than $cf(n)$

Big O puts an *upper bound* on a function.

Big O Examples

- Show that $5n^2 \in O(n^2)$
  - Pick a constant and a value N such that $5n^2 \leq cn^2$ when $n \geq N$
  - This is easy because there is only one term!
  - i.e. since $5n^2 = 5n^2$, $5n^2$ is *always* $\leq 5n^2$. Hence, we can pick 0 for N and 5 for C
    - $5n^2 \leq 5n^2$ for $n \geq 0$

- Show that $[\, n(n - 1)\,] / 2 \in O(n^2)$
  - $[n(n - 1)] / 2$
    $\leq [n(n)] / 2$
    $= \frac{1}{2}n^2$
- Now that we have a single term, we simply pick 0 for N and ½ for C:
  - $\frac{1}{2}n^2 \leq \frac{1}{2}n^2$ for $n \geq 0$

Big O Examples

Note: There are not unique values for c and N that show a function to be in a certain complexity class.

- We previously proved $n^2 + 10n \in O(n^2)$ using $c = 2$ and $\mathrm{N} = 10$
- However, we could use different constants:
  - $n^2 + 10n$
    $\leq n^2 + 10n^2$
    $= 11n^2$

$n^2 + 10n \leq 11n^2$ for $n > 0$. This time we used $c = 11$ and $\mathrm{N} = 0$

Big O Examples

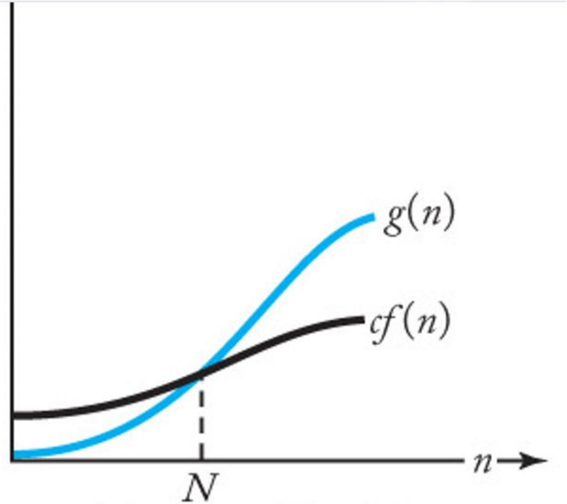
Note: Although $n \in O(n)$, it is also $\in O(n^2)$

- A complexity function doesn't need a quadratic term to be $O(n^2)$.
    - It only needs to eventually lie beneath some quadratic function on a graph.
- Therefore, any complexity function can be placed in a higher big O than the lowest big O it belongs in.
- Example: $n \in O(n^2)$
    - $n \leq 1 \times n^2$ with $c = 1$ and $N = 1$

➢ It is usually *not* useful to place a function into a higher Big O. We want to keep it as tight as possible. $n \in O(n)$ is much more useful.

Omega

- For a complexity function $f(n)$, $\Omega(f(n))$ is the set of all complexity functions $g(n)$ for which there exists some positive real constant c and some nonnegative integer N such that for all $n \geq N$: $g(n) \geq c * f(n)$
- Just as Big O shows an *upper bound* on a function, Omega shows a *lower bound*.



(b) $g(n) \in \Omega(f(n))$

For example

If we can find **some** cubic function that is always as good as or better than g(n) over time, then g(n) $\in \Omega(n^3)$

Omega Example

Show that $n^2 + 10n \in \Omega(n^2)$

- $n^2 + 10n \geq n^2$
  - n^2 is a quadratic function that is always better than $n^2 + 10n$ when $n \geq 0$.
    - Therefore, picking $c = 1$ and $\mathrm{N} = 0$ proves $n^2 + 10n \in \Omega(n^2)$

Theta

Earlier we discussed an intuitive technique for determining the *order* of a function. The more rigorous definition is:

- For a given complexity function f(n), Θ(f(n)) = O(f(n)) ∩ Ω(f(n))

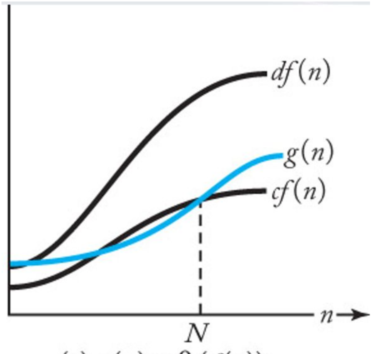
- In other words, all functions that are **both** big O and omega of f(n) are order f(n).

Since we have proven

- $n^2 + 10n \in \Omega(n^2)$ and
- $n^2 + 10n \in O(n^2)$

we know that

- $n^2 + 10n \in \Theta(n^2)$



(c) $g(n) \in \Theta(f(n))$

In general, order is what we will be determining most often in this class.

Order Examples

Suppose a sorting algorithm has a complexity ($n\lg n$)

- If it takes 100 units of time for a list of length 64 to be sorted, how long will it take for a list of length 512 to be sorted?

$64\lg 64$: 100 units = $512\lg 512$: x units

- $64 \times 6 / 100 = 512 \times 9 / x$
  $x(64 \times 6 / 100) = 512 \times 9$
  $x = (512 \times 9 \times 100) / (64 \times 6) = 1200$ units

Order Examples

When given two functions, we can determine if they are in the same complexity class by seeing if they grow at about the same rate with n.

- If both functions can be multiplied by a constant that will cause it to always be larger than the other function, regardless of the size of n, then they are in the same complexity class.

Are $n^2 3^n$ and $n^3 2^n$ in the same complexity class?

Order Examples

Are $n^2 3^n$ and $n^3 2^n$ in the same complexity class?

- Throw away n^2 and n^3 as they are lesser terms. We are left with:
  - 3^n and 2^n
  - Are they in the same complexity class?

Order Examples

Are $n^2 3^n$ and $n^3 2^n$ in the same complexity class?

- Throw away n^2 and n^3 as they are lesser terms. We are left with:
  - 3^n and 2^n
  - Are they in the same complexity class? **No**!
    - 3^n grows much faster than 2^n.
    - There is no constant that we could multiply 2^n by to guarantee it will always be greater than 3^n.

Are 2^n and 2^{n+2} in the same complexity class?

Order Examples

Are $n^2 3^n$ and $n^3 2^n$ in the same complexity class?

- Throw away n^2 and n^3 as they are lesser terms. We are left with:
  - 3^n and 2^n
  - Are they in the same complexity class? **No**!
    - 3^n grows much faster than 2^n.
    - There is no constant that we could multiply 2^n by to guarantee it will always be greater than 3^n.

Are 2^n and 2^{n+2} in the same complexity class?

- $2^{n+2} = 2^2 \times 2^n$
- We are left with 2^n and $2^2 \times 2^n$
- Throw away the constant (2^2) and both sides are 2^n.
- Therefore, they are in the same complexity class!

Order Wrap-Up

- While order is important in algorithm analysis, it is not the *only* important factor.
- Time complexity provides more info overall.

- Imagine two algorithms, a and b:
  - a has a time complexity of $100n$ and b a time complexity of $0.01n^2$
  - Although $b \in \Theta(n^2)$ and $a \in \Theta(n)$, it will take a very large n for b to perform worse than a.
  - To determine *how* large, we find what n causes $0.01n^2 > 100n$ to be true.
    - Divide both sides by $0.01n$. We get $n > 10{,}000$
  - If n will always be less than 10,000, we should use algorithm b!

- While this example is extreme, it's still important to keep this principle in mind.

Order Wrap-Up

Although we will often rigorously prove an algorithm's order, there are some common-sense rules of thumb to determine order.

- If the input is traversed one time, it's most likely $\Theta(n)$
- If the input is traversed in two nested loops, it's likely $\Theta(n^2)$
- If the input is traversed in three nested loops, it's likely $\Theta(n^3)$
- If the input is cut in half each iteration, it's likely $\Theta(\lg n)$
- If the input is traversed one time, and inside that loop, the input is cut in half each iteration, it's likely $\Theta(n \lg n)$
- If our algorithm returns the result of two recursive calls, where the input is *not* cut in half but rather decremented by one each iteration, it's likely $\Theta(2^n)$

In-Class Exercise

1. $f(x) = 3n^2 + 10n\lg n + 1000n + 4\lg n + 9999$ is in which complexity class?

   $\Theta(\lg n)$, $\Theta(n^2\lg n)$, $\Theta(n)$, $\Theta(n\lg n)$, $\Theta(n^2)$

2. Determine whether the following are in the same complexity class:
   a. $2^{\lg n}$ and n
   b. $n^{1/2}$ and $(\lg n)^2$
   c. $(n - 1)!$ and $(n)!$

3. Prove that $6n^2 + 20n \in O(n^2)$