Module 1 Review

Algorithm Analysis

Algorithm Analysis Approach 1 (Simulation: Timing)

```
auto t1 = clock::now()
for(int i = 0; i < 1000; i++);
auto t2 = clock::now();
print t2 - t1;
```

output: 2910 nanoseconds
*output when (i < 1,000,000) = 2761565

Pros:
Easy to measure and interpret.

Cons:
* Results vary across machines.
* compiler dependent
* Results vary across implementations.
* Not predictable 4 small inputs.
* No clear relationship b/w input & time

Approach 2 (Modeling: Counting)

```
int sum = 0;          1
for(int i = 0;        1
    i < n;            n+1
    i++)              n
    sum += i;         n
print sum;            1
```

Simple symbolic count = ▨
$T(n) = 3n + 4$

Pros: independent of computer
Input dependence is captured in model

Cons: No definition of which operation to count
- Tedious to compute *
- Results vary across implementations.
- Doesn't tell actual time.

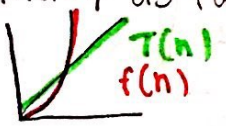
Approach 3 (Asymptotic Behavior: Order of Growth) THE IMPORTANT ONE.

• Order of Growth:

$$1 < \log(n) < n < n\log(n) < n^2 < n^3 < 2^n < n!$$

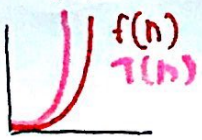
Notations for Algorithm complexity — If you read nothing else, read this...

- If something is $T(n) \in O(f(n))$ it means that $f(n)$ is an UPPER BOUND on the function $T(n)$. * In other words, $T(n)$ grows SLOWER THAN $f(n)$ or equally as fast as $f(n)$



Ex] n is $O(n^2)$ b/c n grows slower than n^2
$T(n) \leq f(n)$

- If something is $T(n) \in \Omega(f(n))$ it means that $f(n)$ is a LOWER BOUND on the function $T(n)$. * In other words $T(n)$ grows FASTER THAN $f(n)$ or equally as fast as $f(n)$



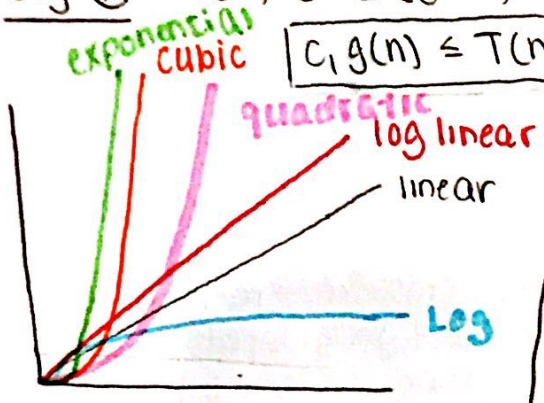
Ex] Which of these functions is $\Omega(n^5 \log_2(n))$
(a) $2n^6$ (b) n^5 (c) $n^5 \log_4(n)$

Answer: (a) and (c) [$2n^6$ grows faster & (c) grows equally as fast.

Algorithm Analysis

- Big Θ: $T(n) \in \Theta(g(n))$ if $T(n) = O(g(n))$ and $T(n) = \Omega(g(n))$

$$c_1 g(n) \leq T(n) \leq c_2 g(n) \text{ for all } n \geq n_0$$



Best / Avg / Worst Case:
- time complexities estimate execution time of algorithm as input → ∞
independent of input size
- measure actual costs @ specific input instance.

Note to self: Learn more about best & worst case

STAY TUNED 4 EXAMPLES

Okay Lets Learn Some Rules:

#1 Addition (independence)

```
for(int i = 0; i < n; i++)
  Not Nested ☺
for(int j = 0; j < m; j++)
```

$T(n) = O(n+m)$ (pretty straightforward)

#2 Drop constant Multipliers

```
for(int i = 0; i < n; i++)
  Not Nested ☺
for(int j = 0; j < n; j++)
```

$T(n) = O(n+n) = O(2n) \sim O(n)$

#3 Different Input Variables

```
function(int n, int l) {
  for(int i = 0; i < n; i++)
    Not Nested ☺
  for(int j = 0; j < l; j++)
```

$T(n) = O(n+l)$
* n and l are different variables!

#4 Drop Lower Order Terms w/ similar Growth rates

```
for(int i = 0; i < n; i++)
  Not Nested ☺
for(int j = 1; j <= m; j*=2)   → log_2 m
```

$T(n) = O(n + \log_2 m) \sim O(n)$

you can ONLY DO THIS when you can assume n and m grow at same rate

Module 1 Problems:

Algorithm Analysis

① True/false: Loglinear functions grow faster than quadratic functions for large inputs.

Solution: false: $n\log n < n^2$ [loglinear grows slower than quadratic functions]

---

② What is time complexity in terms of Big-O for the following

```
for(int i = 1; i < n; i++)
  for(int j=n; j>0; j=j/2)
    print "COP3530";
```

Solution: Work from innermost loop → outward.

$O(n\log(n))$

Notice 1st: The "step" operation in innermost loop ($j=j/2$) is what gives the innermost loop a $O(\log_2 n)$ time complexity.

2nd: The outer for loop will run n times (notice the step operation is simply i++) so the outer loop is $O(n)$

Finally: The loops are nested so we multiply... $n * \log(n)$

---

③ Algorithms total run-time is given by $T(n) = 10n + p$. What is representation of programs execution time in Big-O?

Solution: $O(n+p)$

* There is NO relationship given between n & p.
[like it doesn't say "p & n grow at approximately same rate" or "p grows slower than n"]
↳ so dont go assuming

For Ex] we cannot simplify to $O(n)$ b/c it does not tell us that $p << n$.

Spark Notes: Dont be assuming. Look at exactly what information is given to you.

Lots of words, but if you dont get it, read it!

④ What is time complexity of this code segment?

```
for(int i = 100; i > -1; i--)
  for(int j = i; j > 1; j /= 2)
    print "Hi"
```

Solution O(1). So you might be thinking: ???? But heres whats going on. Start by examining innermost loop. Notice the step operation is $j /= 2$. This indicates we are dealing with logs. But log of what exactly? Note that the start operation in the inner loop is $j = i$. NOT $j = 1$ And what is i? Well look at the outer loop. $i = 100$, so the inner loop (initially) will have time complexity = $O(\log_2 100)$ which is a CONSTANT! what about when i decrements to 99? Well then $j = 99$ (initially) so $O(\log_2 99) \longrightarrow$ ALSO a constant. The way I look at it, the complexity of the inner loop is O(1). The complexity of the outer loop is also O(1) because it will run 100 times. So over all (since the loops are nested, multiply $1 * 1$ and get 1. O(1).

Spark Notes: Read it

⑤ What is time complexity?

```
for(int i = n; i > 0; i /= 2)
  for(int j = 1; j < i; j++)
    sum++;
```

Solution O(n) start with inner loop. The inner loop will run n times b/c the "finish" operation is $j < i$ (so it will run while j is less than i) and i at first is = to n ($i = n$).

Okay but what about when in the outer loop, n decrements to $\frac{n}{2}$. Then the inner loop will run $\frac{n}{2}$ times. next iteration of outer loop? n decrements to $\frac{n}{4}$. inner loop runs $\frac{n}{4}$ times. Starting to see a pattern? Eventually what we will have is this.

$$n + \frac{n}{2} + \frac{n}{4} + \frac{n}{8} \ldots \longrightarrow n\left[1 + \frac{1}{2} + \frac{1}{4} + \ldots\right] \sim O(n).$$

Module 1 Problems: Algorithm Analysis.

⑥ True / false: The best case time complexity for bubble sort is O(1) if the input array only has ONE element.

Solution false: the best / avg / worst case are independent from input size!

⑦ What is computational complexity?

```
// c is positive integer constant
for(int i=1; i<=n; i+=c){
    print "Herro"
}
for(int i=n; i>0; i-=c){
    print "world"
}
```

Solution O(n) So I could be wrong about this (theres NO exact solution posted)

But heres what I think.
the first loop is O(n) because i is just incrementing by a constant # until it reaches unknown variable n.
The 2nd loop is also O(n) b/c it is simply decrementing by constant # starting at unknown var n, and finishing at i > 0. So the whole algorithm is $O(n+n) = O(2n) = O(n)$.

⑧ What is computational complexity?

```
for(int i=1; i<=n; i*=c){
    print "kill"
}
for(int i=n; i>0; i/=c){
    print "me!"
}
// c is constant again.
```

Solution O(log(n)) → No solution posted.

Again correct me if I'm wrong.
But here we go. Top Loop:
The top loop is $O(\log_c n)$ because of the step operation $i*=c$.

The bottom loop is also $(\log_c n)$ because of step: $i/=c$
only difference between the loops is that one is going up to n and one is decrementing down from n. So the whole algo is $O(\log_c(n) + \log_c(n)) = O(2\log_c n) = O(\log_c n)$

All I got 4 ya folks! Good Luck
Δ hmu if you have questions OR corrections.