Module 1

**

How to estimate time complexity

- Use arbitrary running time of code fragments

Asymptotic analysis

- Method used to measure algorithm efficiency

Big O Notation

- Worst case
- Upper bound
- We will concentrate on this

Big Ω Notation

- Best case
- Lower bound

Big Θ Notation

- Average case
- Tight bound
- Not really average case, just more precise

Coding best practices

- Use camelCase
- Descriptive variable names

Growth rates

- If algorithm A requires n^2/2 steps and algorithm B requires 5n+10 steps
- B is better
- Prove by checking n values (make a table)
- A is better with the smallest input values but then B becomes better with big data

Worst case: f(x) = O(g(x)) iff f(x) <= C * g(x)

Best case: f(x) = Ω(g(x)) iff f(x) >= C * g(x)

Average case: $f(x) = \Theta(g(x))$ iff

- $f(x) = O(g(x))$ and $f(x) = \Omega(g(x)$

Examples

- If $f(n) = n^2 + 3n$, can we say $f(n) = O(n^2)$
- Make table
- Yes

Common orders

- O(1)
- Any constant order complexity, not necessarily one
- Any integer/double arithmetic/logic operation
- Accessing a variable or an element in an array
- O(n)
- Linear
- $f(n) = a * n + b$
- a is the slope
- b is the y intersection
- For loops
- Doing something n times
- Sublinear
- sqrt(n)
- log(n)
- $2^3 = 8 \Rightarrow \log_2(8) = 3$
- Most common base with computers is base 10
- Slow growing function
- Log properties
- $\log(xy) = \log x + \log y$
- $\log(x^a) = a \log x$
- $\log_a(n) = \log_b(n) / \log_b(a)$
- If you put a Big O you don’t need to write the base
- If you don’t put Big O you have to put the base
- Common for divide and conquer algorithms
- How many times 1000 can be divided by 2 to get <= 1
- $\log_2(1000)$
- Code this

Recurrence tree

$T(n) = T(n/4) + T(n/2) + n^2$

Make a tree with T(n) at the top

n^2

(n/4)^2 (n/2)^2

T(n/16) T(n/8) T(n/8) T(n/4)

You can keep going on and on

Doing at least 4 levels is good

Horizontally add each level

n^2

5/16 n^2

25/256 n^2

n^2 + 5/16 n^2 + 25/256 n^2 + …

Take n^2 as common and assuming r = 5/16,

n^2 (1 + r + r^2 + r^3 + …)

GP Series (Geometric series)

n^2 (1 - r^k / 1 - r)

If r is less than 1, then you can ignore the summation part

So keep only the n^2 because it is the highest order

O(n^2) is the complexity

T(n) = T(n/2) + C

C

T(n/2)

T(n/4)

T(n/8)

T(n/16)

…

O(1)

n/2 -> n/4 -> n/8 -> n/16

After k times it becomes a constant order

(n/2) ^ k = 1

k = log2(n)

Don’t assume the base

Base is irrelevant only if you write it in Big O form O(log n)

Tree height * complexity at each level

T(n) = 3 T(n/4) + n

n

(n/4) (n/4) (n/4)

T(n/16) T(n/16) T(n/16) T(n/16) T(n/16) T(n/16) T(n/16) T(n/16) T(n/16)

n

3 / 4 n

9 /16 n

n (1 + r + r^2 + …)

O(n)

T(n) = 2 T(n/2) + n

O(n log n)

T(n) = 4 T(n ^ 1/4) + log base 4 (n)

Masters Theorem

T(n) = a T(n / b) + f(n)

(where a <= 1, b < 1, and f is asymptotically positive)

1.

f(n) = O (n ^ (log base b a - ε)) for some constant ε > 0

=> f(n) grows polynomially slower than n ^ (log base b a) by n^ε factor

EXAMPLE

$T (n) = 4 T (n / 2) + n$

$a = 4$

$b = 2$

$f(n) = n$

n : n ^ (log base2 4)

n : n^2

n : n^(2 - ε) = n ^ (2 - 1)

n : n

$\varepsilon > 0$

MT 1

3.

f(n) = Ω(n ^ (log base b a) + ε) for some constant $\varepsilon > 0$

=> f(n) grows polynomially faster than n ^ (log base b a) by n^ε factor

And f(n) satisfies the condition that a f (n / b) <= c f(n) for some constant c < 1

T(n) = O (f(n))

2.

Compare f(n) with n ^ (log base b a)

f(n) and O (n ^ (log base b a) * lg^k n) for some constant k >= 0

=> f(n) and n ^ (log base b n) grow at similar rates

T(n) = O(n ^ (log base b a) *lg ^ (k + 1)* n)

lg is log base 2

**