COSC 2306
Data Programming

Algorithm Analysis

Magnitude of function

- Distinguish the increase speed of function
  - Find the fastest-increasing term in f(n)
  - Discard the constant coefficient
- Constant < Logarithm < Power < Exponential
- Power functions: higher degree is faster
  - $n^{0.5} < n < n^2 < n^3$
- Exponential functions: larger base is faster
  - $2^n < 3^n < 4^n$
- Category first
  - $10^{100} < \log n < n^{0.0001}$
  - $n^{100000} < 1.00001^n$

The Big-O notation

- Big-O notation: describe how an algorithm's running time/memory use grows as the input size n increases with the focus on the dominant growth term
  - Only one term
    - Use $O(n^2)$ instead of $O(n^2+n)$
  - No constant coefficients
    - Use O(logn) instead of O(2logn)

Usage of Big-O

- Why we need Big-O?
- Estimate the efficiency of programs
- Guide the optimization of programs
- For large-scale problems:
  - Exponential Big-O is never acceptable
  - For $O(n^q)$: reduce q as much as possible

Determine Big-O

To find the Big-O notation of a program:

- 1. Find the number of operations f(n)
- 2. Find the fastest increasing term of f(n)
  - $f(n) = n^3 + n^2 = O(n^3)$
- 3. Remove constant coefficients
  - $f(n) = 2^{n+1} = 2*2^n = O(2^n)$

Some examples

Function	Big-O
$f(n) = a$ (a non-negative real)	
$f(n) = 2n + 5$	
$f(n) = n^2 + 3n + 2$	
$f(n) = 4n^6 + 3n^3 + 8$	
$f(n) = 2\log_2 n + a$	
$f(n) = 4\log_2 n + 4n$	
$f(n) = n\log_2 n + 4n$	

Some examples

Function	Big-O
f(n) = a (a non-negative real)	O(1)
f(n) = 2n + 5	O(n)
$f(n) = n^2 + 3n + 2$	$O(n^2)$
$f(n) = 4n^6 + 3n^3 + 8$	$O(n^6)$
$f(n) = 2\log_2 n + a$	$O(\log_2 n)$
$f(n) = 4\log_2 n + 4n$	O(n)
$f(n) = n\log_2 n + 4n$	$O(n\log_2 n)$

Exercise

- Put in order by big-O bound

$4n^2$	$\log_3 n$	$20n$	2	$\log_2 n$	n^n	3^n	$n\log n$	2^n	2^{n+1}

Exercise

- Put in order by big-O bound

$4n^2$	$\log_3 n$	$20n$	2	$\log_2 n$	n^n	3^n	$n\log n$	2^n	2^{n+1}

2	$\log_2 n = \log_3 n$	$20n$	$n\log n$	$4n^2$	$2^n = 2^{n+1}$	3^n	n^n

Determine Big-O

- Number of operations in programs
  - Count arithmetic and other more complicated operations (a+b, a*b)
  - Assignments (a=b) and comparison (a<b) is usually count
  - For loops:
    - Num of operations = num of loop round * operations per loop
    - Nested loop:
      - for … # n rounds
      - for … # m rounds
      - In total n*m rounds.
      - Other cases: Do mathematics to count the rounds.

Determine Big-O

```
i = n
while i > 0:
        m = 2 + 2
        i = i // 2
```

Assignment for i is 1 op
For each i, two statements result in 2 ops,
i is floor divided by 2, so i = n, n/2, $n/2^2$, $n/2^3$,
…, $n/2^k$, k+1 steps until i = 0 or $n/2^k = 1$ -> $n = 2^k$
-> logn = $\log 2^k = k$, so takes logn + 1 iterations
f(n) = 1 + logn + 1
O(logn)

```
result = 0
for i in range(n):
        for j in range(n):
                result = i*j
                print(result)
```

Assignment for result is 1 op
Each i results in n loops of j
For each i, j pair, two statements result in 2,
so the nested for loop is $2n^2$
$f(n) = 1 + 2n^2$
$O(n^2)$

Determine Big-O

```
result = 0
for i in range(n):
        if result % 2 == 0:
                for j in range(n):
                        result+=1;
        else:
                print(result)
```

Initially result = 0, which is **even**, so the first if is true
That means the **first iteration** of the outer loop executes the **inner loop**:
- Inner loop executes n times, each time incrementing result by 1
- After the first inner loop finishes, result = n (started at 0 and incremented n times)

Now, result = n
- If **n is even**, then result % 2 == 0 again, so the inner loop runs on **every iteration**
- If **n is odd**, then result % 2 == 1, so the else branch executes, printing once. On the next iteration, result is still odd → print again, and so on. No inner loops after the first run

Case 1: n is even: every outer iteration runs the inner loop of size n → **$O(n \times n) = O(n^2)$**
Case 2: n is odd: first iteration runs the inner loop (O(n)), then the rest just print (O(1) each, repeated n-1 times) → **$O(n + (n-1)) = O(n)$**