Reminders

- Project 4 is due this Friday 3:59pm.
- This Lecture: Algorithm Analysis (Big-O notation)

What is Algorithm Analysis?

- Resources (time, memory etc.) used by an algorithm, as a function of the input size (a.k.a. problem size).
- The cost may be different for different inputs of the same size -- we often consider the worst-case cost because we want to make a guarantee to the user.
- We will focus on analyzing the time complexity of an algorithm, in terms of the worst-case running time (e.g. number of instructions).

Problem A: calculate the sum of the numbers in an array with n elements.

```
double sum = 0.0;
int n = A.length;
for (int i=0; i < n; i++) {
    sum += A[i];
}
return sum;

2 assignment

n iterations

1 return value</pre>
```

Problem A: calculate the sum of the numbers in an array with n elements.

Total number of instructions or steps:

n + 3

We call this **linear** w.r.t. the problem size n. If the array has 3 times as many elements (i.e. n is 3 times as large), the algorithm will take roughly 3 times as long to run. So the computation cost grows **linearly** with respect to n (assuming n is large).

Problem B: given an array of n numbers, calculate the sum of the even-indexed elements.

```
double sum = 0.0;
int n = A.length;
for (int i=0; i < n; i+=2) {
    sum += A[i];
}
return sum;</pre>
```

Problem B: given an array of n numbers, calculate the sum of the even-indexed elements.

Total number of instructions / steps: (n/2) + 3

This is still **linear** w.r.t. the problem size n. For example, if n is 3 times as large, the run time will be roughly 3 times as long. What we care about is not the precise running time, but rather, **how the running time grows or scales as n increases.**

Clicker Question #1

Which of these loops takes **constant** time, regardless of the value of n (assuming n is a large positive integer)?

Example: Double Loop

Problem C: given an array of n numbers, calculate the sum of all **pairwise multiplications**.

```
double sum = 0.0;
int n = A.length;
for (int i=0; i < n; i++) {
    for (int j=0; j < n; j++) {
        sum += A[i]*A[j];
    }
}
return sum;</pre>
```

Example: Double Loop

Problem C: given an array of n numbers, calculate the sum of all **pairwise multiplications**.

Total number of instructions / steps:

$$n*n + 3 = n^2 + 3$$

This is **no longer linear** w.r.t. the problem size n! As n becomes 3 times as large, the algorithm takes 9 times as long to run. This is a **quadratic** increase, and it grows more rapidly than linear.

Big-O Notation

A notation that expresses computation time (complexity) as the term in the cost function that increases the most rapidly relative to the problem size.

- O stands for 'order', as in 'order of magnitude'.
- We assume n is sufficiently large (towards infinity), hence we only care about the fastest growing term (i.e. highest order term, or the dominant term).
- Constant scaling factors do not matter as it does not affect the rate of growth.
- Just count the number of instructions, no need to think about the relative cost of different operations.

Big-O Example

- n + 3 -> O(n)
- (n/2) + 3 -> O(n)
- $n^2 + 3 -> O(n^2)$
- Imagine an algorithm running on an n-element array requires $f(n) = 2n^2 + 4n + 3$ instructions.
 - The fastest growing term is 2n²
 - The constant 2 in 2n² can be ignored.
- So the time complexity of the algorithm is O(n²).

Order of Terms

- If we graph 0.0001n² against 10000n, the linear term would be larger for a long time, but the quadratic one would eventually catch up (here at n = 10⁸).
- In calculus we know that

$$\lim_{n \to \infty} \frac{10000 \, n}{0.0001 \, n^2} = \lim_{n \to \infty} \frac{10^8}{n} = 0$$

 As you can see, any quadratic (with a positive leading coefficient) will eventually beat any linear. So the linear term in a quadratic function eventually does not matter.

Order of Terms

• Consider the function $n^4 + 100n^2 + 500 = O(n^4)$

	And the second s	•	*	
n	n ⁴	100n ²	500	f(n)
1	1	100	500	601
10	10,000	10,000	500	20,500
100	100,000,000	1,000,000	500	101,000,500
1000	1,000,000,000	100,000,000	500	1,000,100,000,500

 The growth of a polynomial in n, as n increases, depends primarily on the degree (the highest order term), and not on the leading constant or the low-order terms.

Big-O Summary

- Write down the cost function (i.e. number of instructions in terms of the problem size n)
 - Specifically, focus on the loops and find out how many iterations the loops run
- Find the highest order term
- Ignore the constant scaling factor.
- Now you have a Big-O notation.

Example: Double Loop

Problem D: given n numbers in an array A, calculate the sum of all **distinct** pairwise multiplications.

```
double sum = 0.0;
int n = A.length;
for (int i=0; i < n; i++) {
    for (int j=i; j < n; j++) {
        sum += A[i]*A[j];
    }
}
return sum;</pre>
How many times
does this instruction run?
```

Example: Double Loop

Problem D: given n numbers in an array A, calculate the sum of all **distinct** pairwise multiplications.

```
double sum = 0.0;
int n = A.length;
for (int i=0; i < n; i++) {
    for (int j=i; j < n; j++) {
        sum += A[i]*A[j];
    }
}
return sum;</pre>
```

$$n + (n - 1) + (n - 2) + \dots + 2 + 1 = \frac{n(n + 1)}{2} = O(n^2)$$

Clicker Question #2

```
int count = 0;
for (int i=1; i < n; i*=2) {
    count++;
}</pre>
```

What's the cost in Big-O notation?

- (a) O(n)
- (b) $O(n^{1/2})$ (i.e. square root of n)
- (c) $O(2^n)$
- (d) $O(n^2)$
- (e) $O(log_2n)$

Logarithmic Cost O(log n)

```
for(int i=1; i < n; i*=2) {...}
for(int i=1; i < n; i<<=1) {...}
for(int i=n; i>0; i/=3) {...}
for(int i=n; i>0; i>>=2) {...}
base 2
base 2
base 3
```

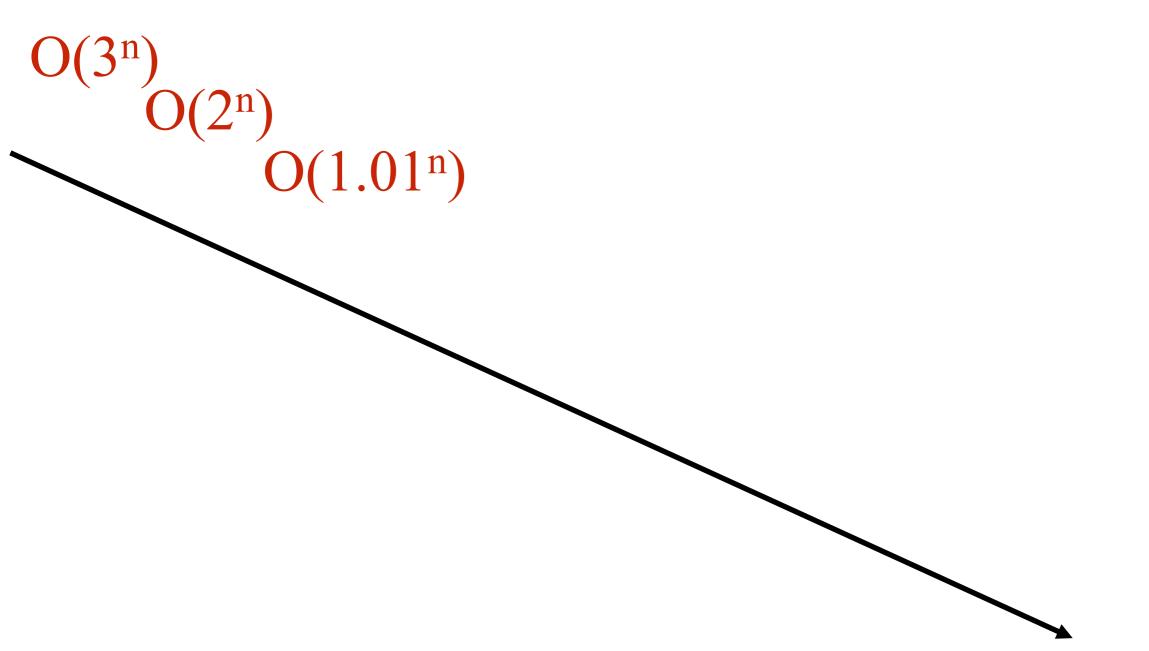
The base does not matter, because:

$$O(\log_2 n) = O(\ln n)/O(\ln 2) = O(\ln n)$$

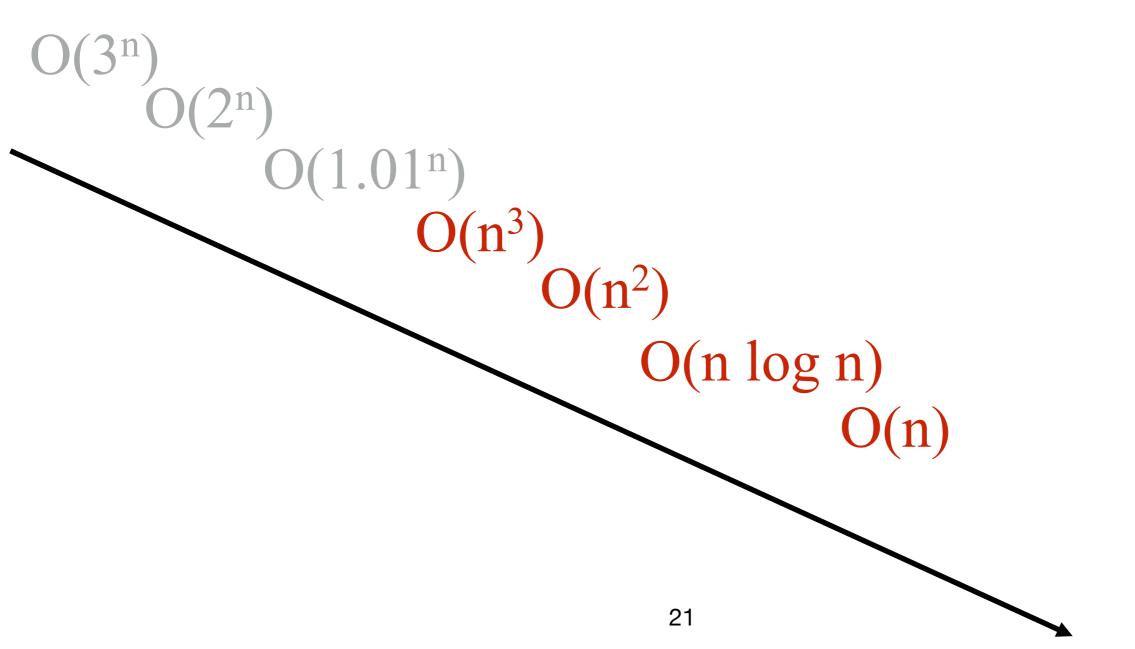
Change of base Base e (natural log)

From calculus, we know that in terms of order:

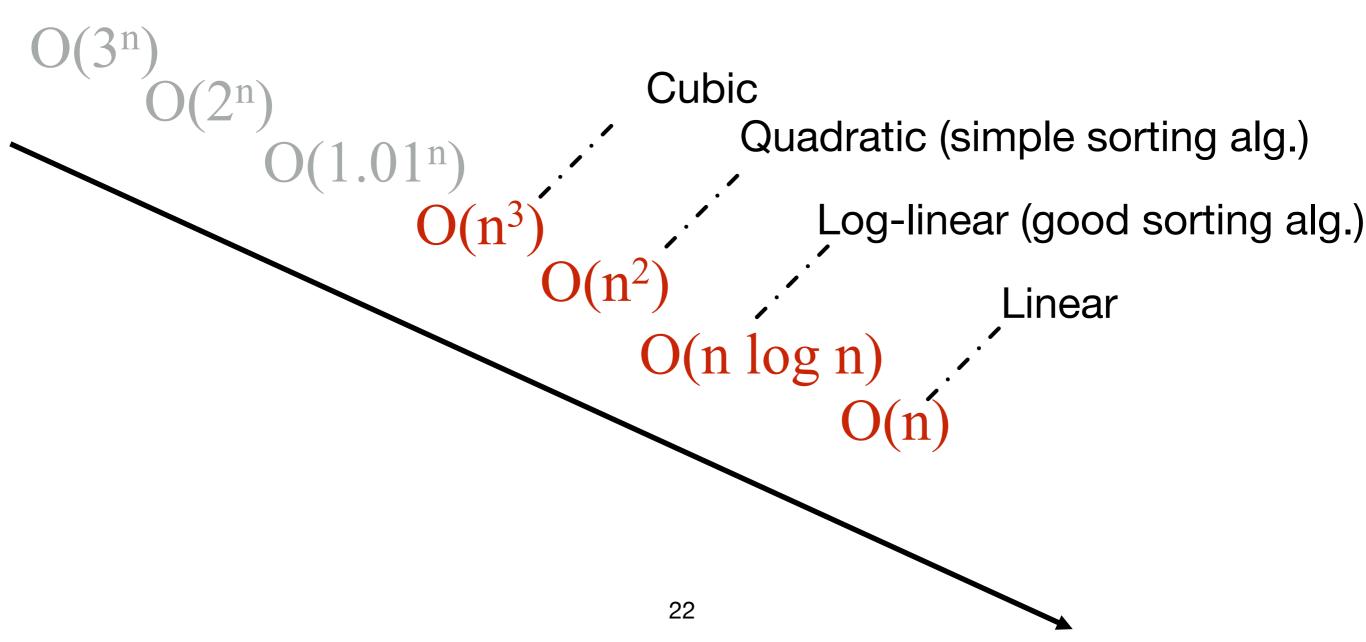
From calculus, we know that in terms of order:



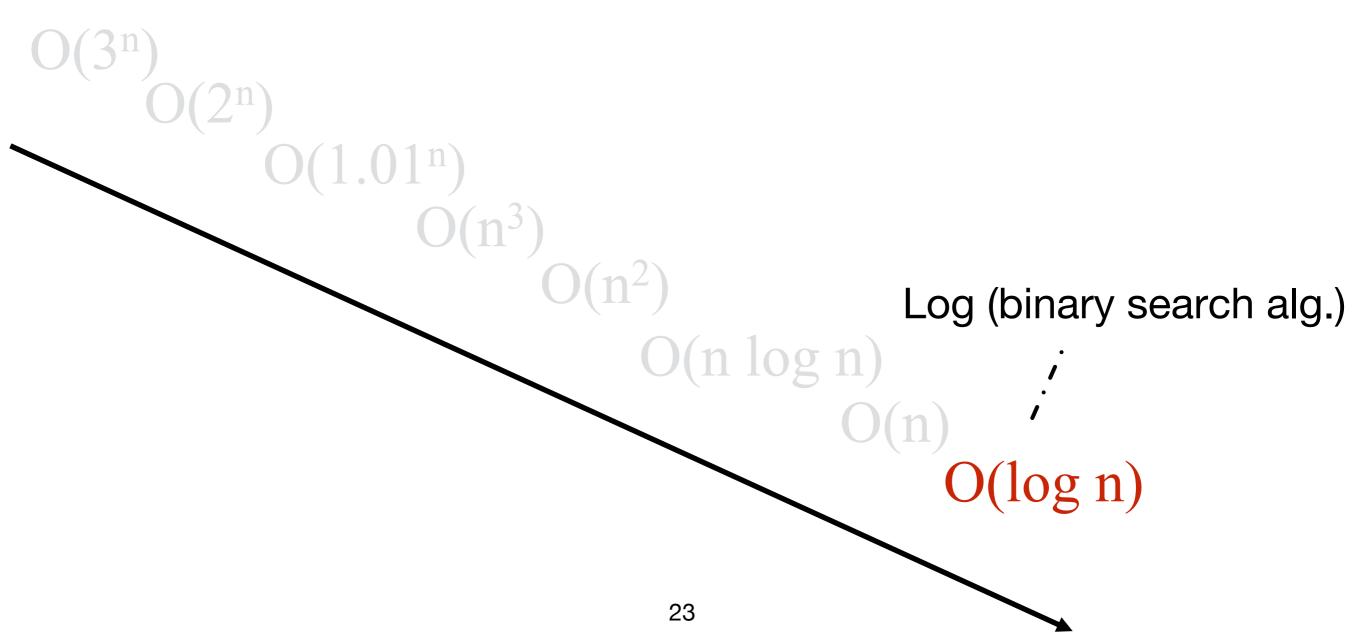
From calculus, we know that in terms of order:



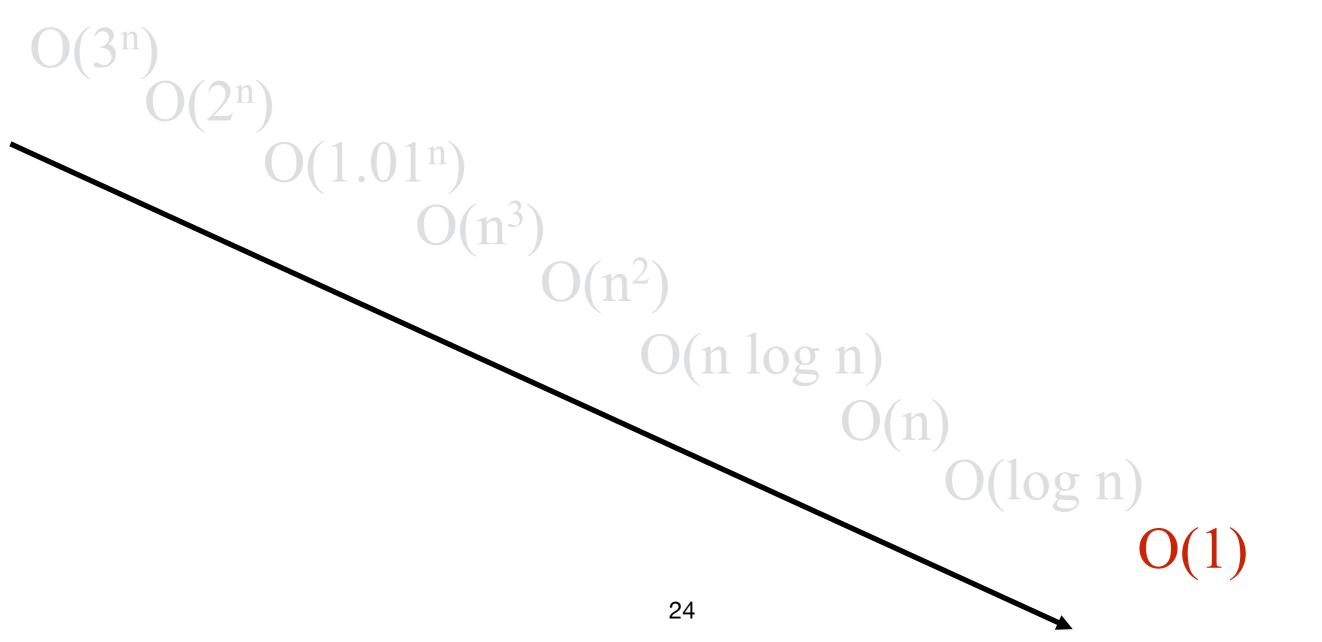
From calculus, we know that in terms of order:



From calculus, we know that in terms of order:



From calculus, we know that in terms of order:



From calculus, we know that in terms of order:

- Look at how doubling n affects each running time:
 - For a constant function, there is no change.
 - For a log function, it increases slightly.
 - For a linear function, the running time doubles.
 - For a quadratic function, it multiplies by four.
 - For exponential, it squares!

Clicker Question #3

For sufficiently large n, which of these four functions will have the largest value?

$$a(n) = n + 100$$

 $b(n) = n^5 + log(n) - n^4$
 $c(n) = 10*n^4 + 100*n + 1000$
 $d(n) = 100*n^4*log(n)$

Clicker Question #4 (tricky)

```
int i, j, count=0;
for (i=1; i <= n; i*=2) {
    for (j=i; j > 0; j--) {
        count ++;
    }
}
```

What's the cost in Big-O notation?

```
(a) O(n)
(b) O(log n)
(c) O(n log n)
(d) O(n²)
(e) O((log n)²)
```

Clicker Question #4 (cont.)

```
int i, j, count=0;
for (i=1; i <= n; i*=2) {
    for (j=i; j > 0; j--) {
        count ++;
    }
}
```

- First, think about some concrete examples:
 when n is 16, number of iterations is 1+2+4+8+16 = 31
 when n is 32, number of iterations is 1+2+4+8+16+32 = 63
- Observe the pattern, we can see that the number of iterations is 2n-1 or O(n)