CS280 – Data Structures

Introductory Algorithm Analysis

What is an algorithm?

Algorithm

- Any well-defined computational procedure that transforms some inputs into some outputs.
- Computer independent
- Programming language independent

Example

- Searching
  - Input: A sequence of n numbers $\{a_1,a_2,\ldots,a_n\}$ and a number k
  - Output: true if k is found in the sequence and false otherwise
- Sorting
  - Input: A sequence of n numbers $\{a_1,a_2,\ldots,a_n\}$
  - Output: A permutation $\{a_1',a_2',\ldots,a_n'\}$ of the input sequence such that $a_1' \leq a_2' \leq \ldots \leq a_n'$

Algorithm Analysis

- Correctness analysis
- Complexity analysis

Algorithm Analysis

- Correctness analysis
- **Complexity analysis**

What is this for?

- The point of **algorithm complexity analysis** is to be able to say that one algorithm is **better** than the other.
  - What does it mean to be better?
  - How to quantify it?

Search Example 1

- Assume we have an array of **random** integers.
- We want to find out where x is in the array.

```
// Assume a declaration like: int a[SIZE];
int n = 0;
while (x != a[n])
  ++n;
```

- What is the least number of iterations?
- What is the most number of iterations?
- What is the average number of iterations?
- What **search method** did you use to arrive at these numbers?

Linear Search

Linear Search

```
int LinearSearch(int *array, int size, int value){
  //Assumption: value does exist in the array
  int i=0;
  while (value != array[i])
     ++i;
  return i + 1;
}
// Assume a is unsorted array of integers of size SIZE = 10000
// Search for random numbers (10 sets)
for (int j = 0; j < 10; ++j){
  int total = 0;

  int attempts = 1000;
  for (int i = 0; i < attempts; ++i)
     total += LinearSearch(a, SIZE, (rand() % SIZE));

  cout<<(j+1)<< ". Average = "<<(double)total/(double)attempts<<
   endl;
}
```

Results for SIZE=10,000

1. Average = 5115.57
2. Average = 5047.94
3. Average = 4915.73
4. Average = 4911.44
5. Average = 4856.43
6. Average = 4920.65
7. Average = 4910.12
8. Average = 4841.81
9. Average = 4860.79
10. Average = 4913.42

Bonus!

- How would you generate an <u>unsorted</u> array of <u>unique</u> integers?

Shuffle!

- Create sorted array and then shuffle it!

```
void Shuffle(int *array, int size){
    for (int i = 0; i < size; ++i){
        int r = rand() % size;
        int t = array[i];
        array[i] = array[r];
        array[r] = t;
    }
}
```

```
// Generate an array of unique integers
for (int i = 0; i < SIZE; ++i)
   a[i] = i;
// Mix it up
Shuffle(a, SIZE);
```

Search Example 2

- Suppose the array was sorted.

1	2	3	4	5	6	7	8	9	10

- What search method would you use?

Binary Search

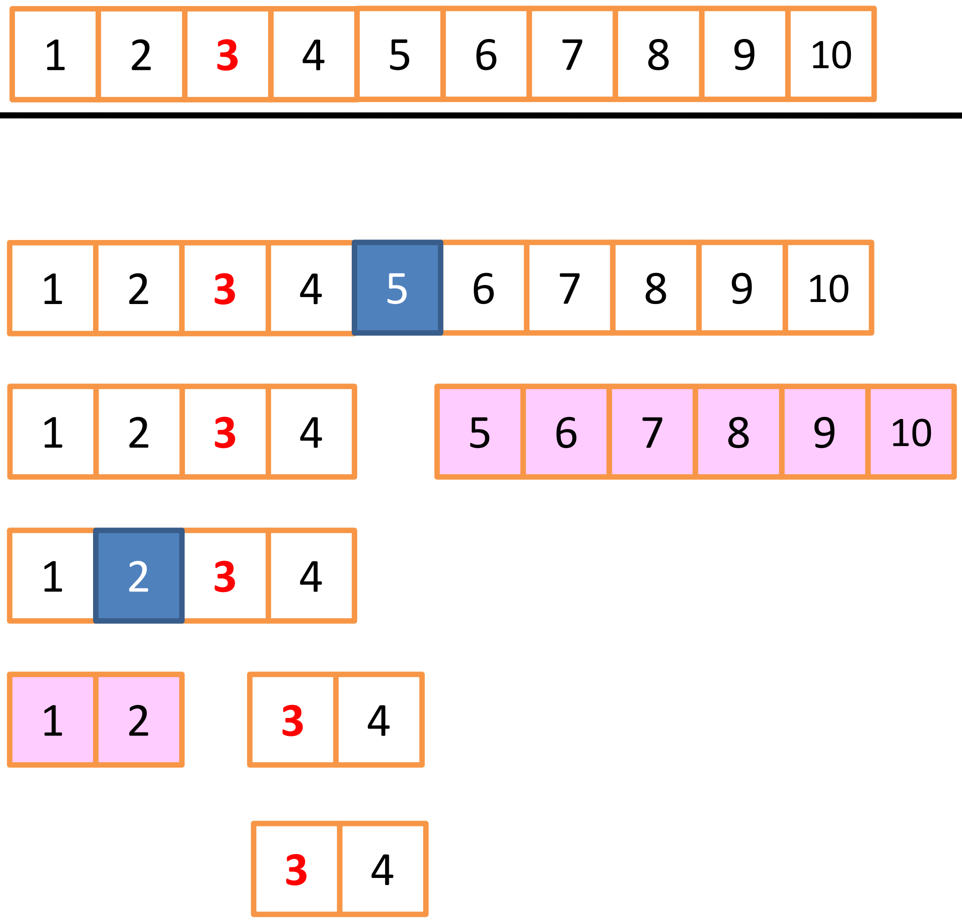
Binary Search

```
int BinarySearch(int *array, int size, int value){
    if (size <= 1)
        return 1;

    int count = 0; // record the number of iterations
    int left = 0, right = size - 1;
    while (right >= left){
        count++;
        int middle = (left + right) / 2;
        if (value == array[middle])
            return count;

        if (value < array[middle])
            right = middle - 1;
        else
            left = middle + 1;
    }
    return count;
}
```

Example



Sorted array

<u>We're looking for 3</u>

Middle index = (0 + 9)/2 = 4

Discard right part of array

Middle index = (0 + 3)/2 = 1

Discard left part of array

Middle index=(2+3)/2=2

Done!

Binary Search

- What is the least number of iterations?
- What is the most number of iterations?

Results for SIZE=10,000

1. Average = 13.51
2. Average = 13.492
3. Average = 13.501
4. Average = 13.483
5. Average = 13.46
6. Average = 13.445
7. Average = 13.517
8. Average = 13.451
9. Average = 13.465
10. Average = 13.516

```
for (int j = 0; j < 10; ++j){
  int total = 0;
  int attempts = 1000;
  for (int i = 0; i < attempts; ++i)
     total += BinarySearch(a, SIZE,
(rand() % SIZE));
  cout<<(j+1)<< ". Average =
"<<(double)total/(double)attempts<<
endl;
}
```

Remember

- The simple search method is called a linear-time algorithm
  - The time is directly proportional to size of the problem
- Binary search is logarithmic-time algorithm
  - The time is proportional to <u>the logarithm of the size of the problem</u>
  - What property must the array have to use a binary search method?

Analysis Points

- Informally, we want to figure out the number of steps required to perform a computation.
- The goal is to write a formula for the computation time :

In terms of the size of the problem

Constants **Variables**

The Big-*O*h Notation

- O(): Worst case asymptotic time complexity

Number of elements	Linear Search	Binary Search
10	10	4
100	100	7
1,000	1,000	10
10,000	10,000	14
100,000	100,000	17
1,000,000	1,000,000	20

Example

- Two algorithms whose running times are described as $N^2/4$ and $200+N\times\log_2 N$
- The computer is able to execute 10^6 instructions/s

N	$N^2/4$	$200+N\times\log_2 N$
10		
100		
1000		
10 000		
100 000		
1 000 000		

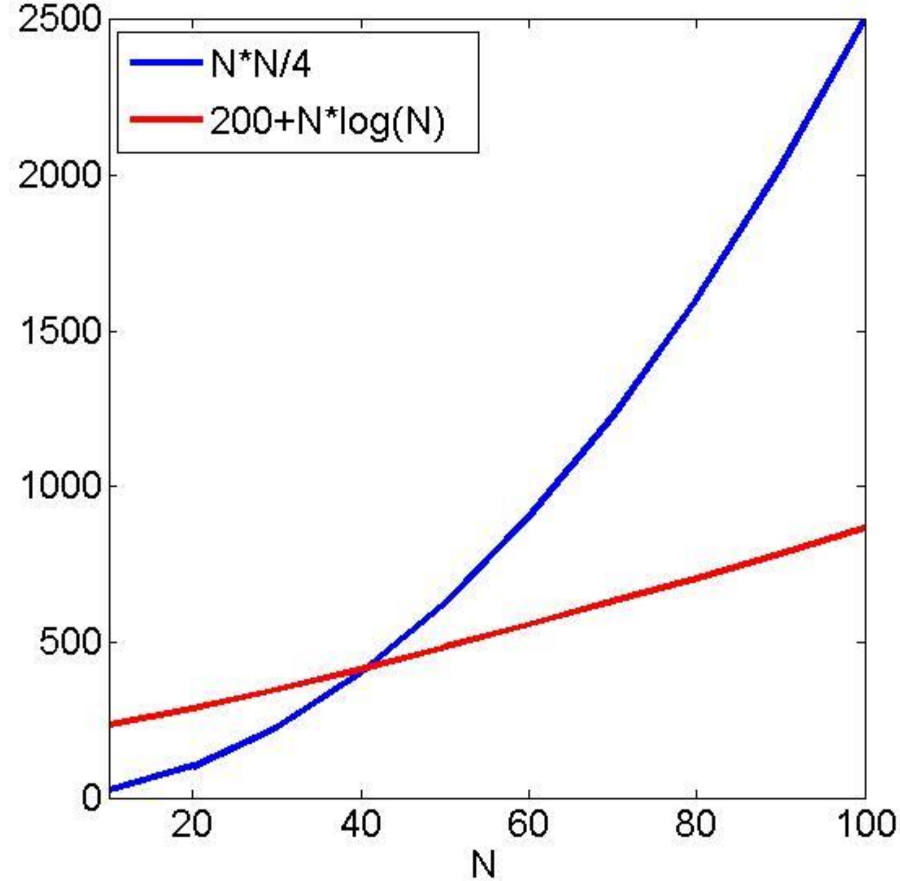
Example

- Two algorithms whose running times are described as $N^2/4$ and $200+N\times\log_2 N$
- The computer is able to execute 10^6 instructions/s

N	$N^2/4$	$200+N\times\log_2 N$
10	**25 μs***	233 μs
100	2.5 ms**	**864 μs**
1000	0.25 sec	**0.01 sec**
10 000	25 secs	**0.13 sec**
100 000	41.67 mins	**1.66 sec**
1 000 000	2.89 days	**19.93 sec**

*μs: microseconds, 1 μs = 10^{-6} second; **ms: millisecond, 1 ms = 10^{-3} second

Example



Dominant Term

- In the **O()** notation, we only care about the **dominant term.**
- In other words, we only care about the term that will account for the **biggest portion** of the running time.

Dominant Term

- We analyze both varying terms: n^2 and $2n$ separately
- $f(n)=n^2+2n+100$

n	f(n)	n^2	n^2 as % of total	2n	2n as % of total
10	220	100	45.455%	20	9.091%
100	10,300	10,000	**97.087%**	200	1.942%
1,000	1,002,100	1,000,000	**99.790%**	2,000	0.2%
10,000	100,020 ,100	100,000,000	**99.980%**	20,000	0.02%
100,000	10,000,200,100	10,000,000,000	**99.99%**	200,000	0.002%

Dominant Term

- Now let’s add a **cubic** term: $f(n)=n^3+n^2+2n+100$

n	f(n)	n^3	n^3 as % of total
10	1,220	1,000	81.967%
100	1,010,300	1,000,000	**97.980%**
1, 000	1,001,002,100	1,000,000,000	**99.890%**
10, 000	1,000,100,020,100	1,000,000,000,000	**99.989%**
100, 000	1,000,010,000,200,100	1,000,000,000,000,000	**99.99%**

Dominant Term

- Now let's add a **exponential** term:
  $f(n)=2^n+n^3+n^2+2n+100$

n	f(n)	2^n	2^n as % of total
10	2,244	1,024	45.632799%
20	1,057,116	1,048,576	**99.192142%**
30	1,073,769,884	1,073,741,824	**99.997387%**
40	1,099,511,693,556	1,099,511,627,776	**99.999994%**

Big-*O*h Notation

- The Big-*O*h Notation
  - An upper bound for complexity of an algorithm
- Formally, $f(n)$'s complexity is $O(g(n))$ means: $\exists n_0 > 0, c > 0$, such that $\forall n \geq n_0$, $0 \leq f(n) \leq c \times g(n)$
- In other words, an algorithm's complexity is $O(g(n))$ means that there exists positive constant c and n_0 whereby when the problem size is greater than n_0, the time required by the algorithm to run is always less than $c \times g(n)$.

Tight Big-*O*h Bounds

- $f(n) = 8n + 128$
- So is $f(n)$ in $O(n)$ or $O(n^2)$?
  - Choose the tighter bound!
- Since n is in $O(n^2)$
  - $O(n)$ is a tighter bound for $f(n)$ than $O(n^2)$
  - <u>“$f(n)$ is in $O(n)$” is a more accurate analysis</u>

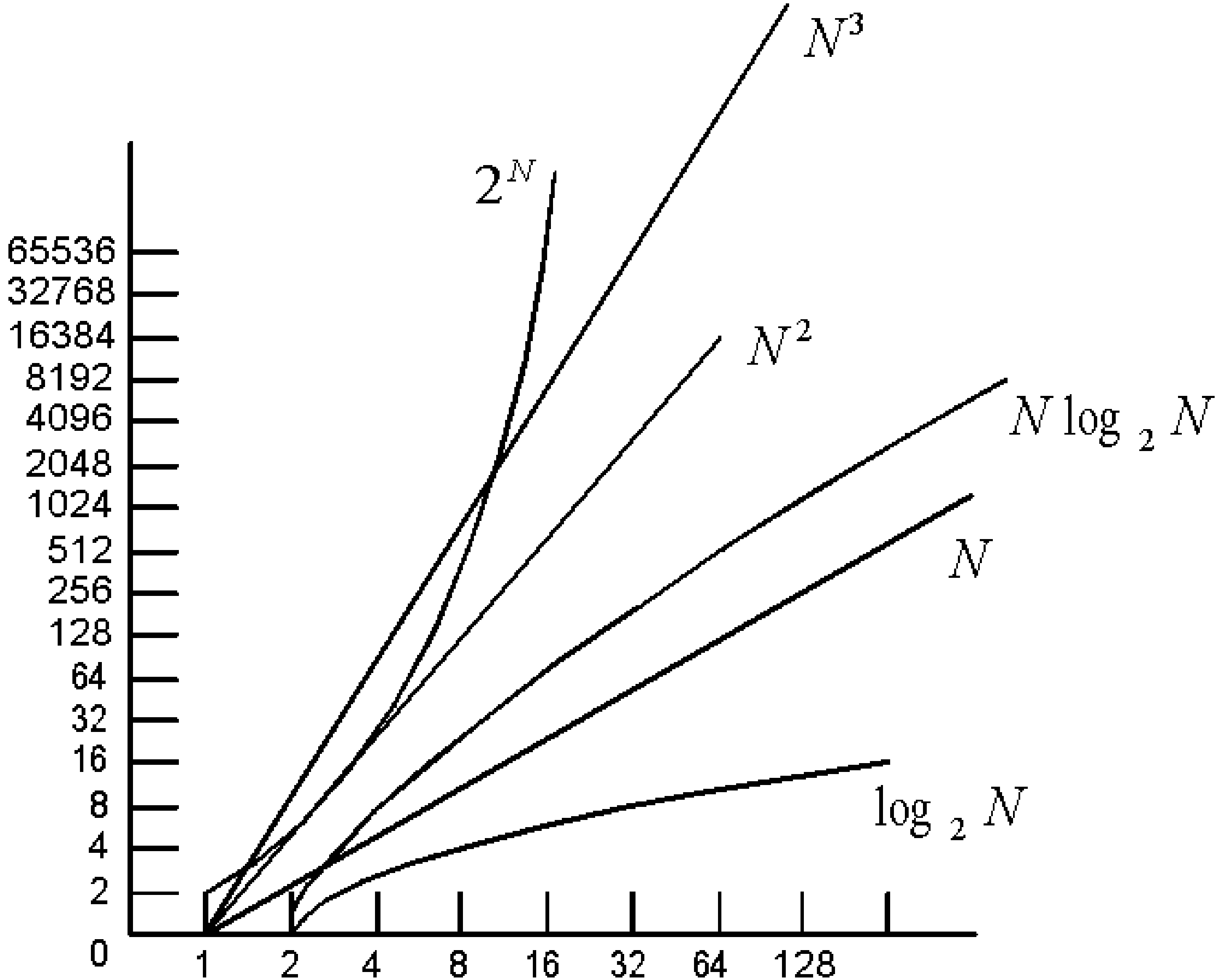
Writing Big-*O*h Expressions

1. Determine running time
   - $n^2+(n \log_2 n)+3n$
2. Drop all but the most significant terms
   - $O(n^2+n\log_2 n+3n) \Rightarrow O(n^2)$
   - $O(n \log_2 n+3n) \Rightarrow O(n \log_2 n)$
3. Drop constant coefficients
   - $O(3n) \Rightarrow O(n)$
   - $O(10) \Rightarrow O(1)$

Common Growth Rates

Growth rate	Name
$O(k)$	Constant
$O(\log_2 N)$	Logarithm
$O(N)$	Linear(directly proportional to N)
$O(N \log_2 N)$	No formal name “N log N”
$O(N^2)$	Quadratic (proportional to square of N)
$O(N^3)$	Cubic (proportional to cube of N)
$O(N^k)$	Polynomial (proportional to N to the power of K)
$O(a^N)(a>1)$	Exponential (proportional to 2 to the power of N)

Common Growth Rates



Common Growth Rates

$\log_2 N$	$(\log_2 N)^2$	$\sqrt{N}$	N	$N\log_2 N$	$N(\log_2 N)^2$	$N\sqrt{N}$	N^2
3	9	3	10	30	90	30	100
6	36	10	100	60	3,600	1,000	10,000
9	8	31	1,000	9,000	81,000	31,000	1,000,000
13	169	100	10,000	1,300,000	1,690,000	1,000,000	100,000,000
16	256	316	100,000	1,600,000	25,600,000	31,600,000	10 billion
19	361	1,000	1,000,000	19,000,000	361,000,000	1 billion	1 trillion

Common Big-*O*h Expressions

Slower

Expression	*Name*
$O(1)$	*Constant*
$O(\log n)$	Logarithmic
$O(n)$	Linear
$O(n \log n)$	n log n
$O(n^2)$	Quadratic
$O(n^3)$	Cubic
$O(n^k)$	Polynomial
$O(2^n)$	Exponential

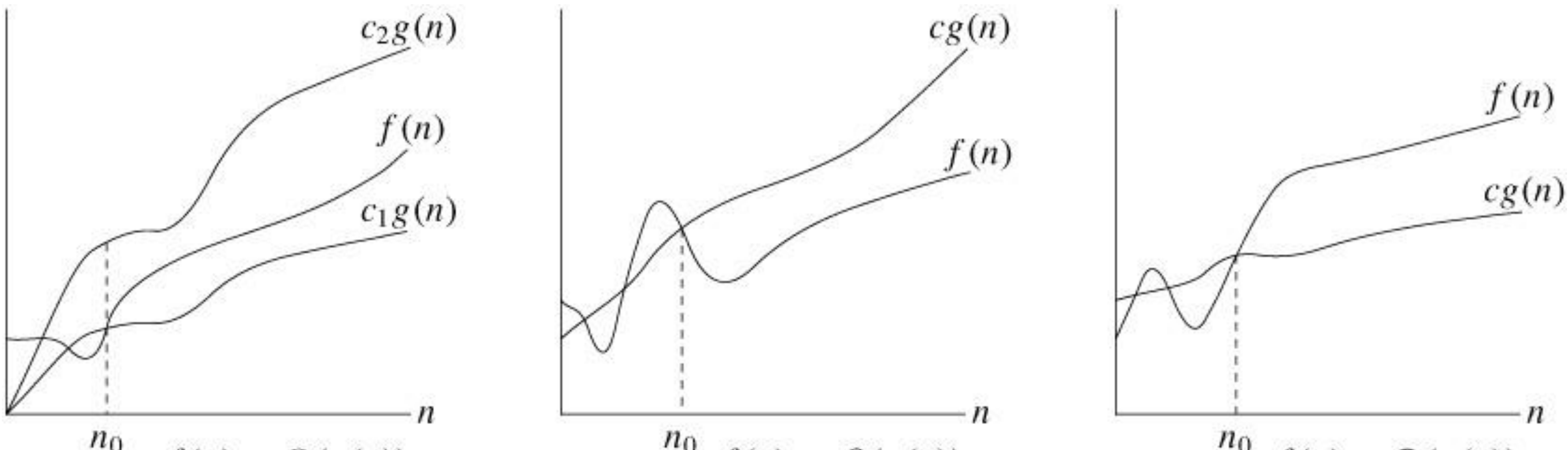
Big-Ω Notation

- The Big-Ω Notation
  - An lower bound for complexity of an algorithm
- Formally, $f(n)$'s complexity is $\Omega(g(n))$ means: $\exists n_0 > 0, c > 0$, such that $\forall n \geq n_0$, $f(n) \geq c \times g(n) \geq 0$
- In other words, an algorithm's complexity is $\Omega(g(n))$ means that there exists positive constant c and n_0 whereby when the problem size is greater than n_0, the time required by the algorithm to run is always more than $c \times g(n)$.

Θ Notation

- Θ notation: the asymptotic <u>tight</u> bound of an algorithm
- Formally, $f(n)$'s complexity is $\Theta(g(n))$ means: $\exists n_0>0, c_1>0, c_2>0$, such that $\forall n \geq n_0$, $c_1 \times g(n) \geq f(n) \geq c_2 \times g(n) \geq 0$
- $f(n)=\Theta(g(n))$ if and only if $f(n)=O(g(n))$ **and** $\mathbf{f(n)=\Omega(g(n))}$

Big-Theta, Big-Oh, Big-Omega



$f(n) = \Theta(g(n))$
(a)

$f(n) = O(g(n))$
(b)

$f(n) = \Omega(g(n))$
(c)

Performance V.S. Complexity

- Performance:
  - Time, memory, disk,
  - Machine, compiler, code
- Complexity:
  - Time complexity: big-O.

Estimating the Growth Rate

- Constant time elementary operations
  - one arithmetic operation (e.g., +, *).
  - one assignment
  - one test (e.g., x == 0)
  - one read
  - one write (of a primitive type)
  - …
  - $T(n) = a \neq f(n)$

Estimating the Growth Rate

- Sequences
- Conditionals
- Loops (this is the big-ticket item)
- Function calls

Sequences

```
Sequence

statement 1;
statement 2;
...
statement k;
```

```
total time =

  T(statement 1)
+ T(statement 2)
+  ...
+ T(statement k)
```

Conditionals

```
Total time = max(T(sequence 1), T(sequence 2))
```

```
if (condition) {
sequence of statements 1
}
else {
sequence of statements 2
}
```

Loops

Total time $=N \times T(\text{statements})$

```
for (i = 0; i < N; ++i) {
sequence of statements
}
```

Nested Loops

```
Total time = N×N×T(statements)
```

```
for (i = 0; i < N; ++i) {
   for (j = 0; j < N; ++j) {
      sequence of statements
       }
     }
```

Function Calls

```
f(n); // O(1)
g(n); // O(n)

for (j = 0; j < N; ++j)
g(j);
```

total time = $O(N^2)$

Summary

- Algorithm complexity analysis
- Big *O*h notation
  - Asymptotic analysis
  - Focus on the dominant term in the expression for running time of your algorithm