CSE 332 Winter 2026
Lecture 3: Algorithm Analysis pt.2

Nathan Brunelle

http://www.cs.uw.edu/332

Running Time Analysis

- Units of “time”
  - Operations
    - Whichever operations we pick
- How do we express running time?
  - Function
    - Domain (input): size of the input
    - Range: count of operations

Defining your running time function

- Worst-case complexity:
  - max number of steps algorithm takes on “most challenging” input
- Best-case complexity:
  - min number of steps algorithm takes on “easiest” input
- Average/expected complexity:
  - avg number of steps algorithm takes on random inputs (context-dependent)
- Amortized complexity:
  - max total number of steps algorithm takes on M “most challenging” consecutive inputs, divided by M (i.e., divide the max total sum by M).

Worst Case Running Time Analysis

- If an algorithm has a worst case **running time** of $f(n)$
  - Among all possible size-n inputs, the "worst" one will do $f(n)$ "**operations**"
  - $f(n)$ gives the maximum count of **operations** needed from among all inputs of size n

Worst Case Running Time – General Guide

- Add together the time of consecutive statements
- Loops: Sum up the time required through each iteration of the loop
  - If each takes the same time, then [time per loop $\times$ number of iterations]
- Conditionals: Sum together the time to check the condition and time of the slowest branch
- Function Calls: Time of the function’s body
- Recursion: Solve a recurrence relation

Worst Case Running Time - Example

```
myFunction(List n){
   b = 55 + 5;
   c = b / 3;
   b = c + 100;
   for (i = 0; i < n.size(); i++) {
      b++;
   }
   if (b % 2 == 0) {
      c++;
   }
   else {
      for (i = 0; i < n.size(); i++) {
         c++;
      }
   }
   return c;
}
```

Questions to ask:

- What are the units of the input size?
- What are the operations we're counting?
- For each line:
  - How many times will it run?
  - How long does it take to run?
  - Does this change with different inputs?
- Answer:

Worst Case Running Time - Example

```
myFunction(List n){
   b = 55 + 5; // 1
   c = b / 3;  // 1
   b = c + 100;  // 1
   for (i = 0; i < n.size(); i++) { // 1, n times
      b++; // 1
  }
   if (b % 2 == 0) { // 1
      c++; // 1
   }
   else {
      for (i = 0; i < n.size(); i++) { // 1, n times
         c++; // 1
      }
   }
   return c;
}
```

Questions to ask:

- What are the units of the input size?
  - # of items in the list
- What are the operations we're counting?
  - Arithmetic ops (+-*/)
- For each line:
  - How many times will it run?
  - How long does it take to run?
  - Does this change with different inputs?
- Answer:
  - $3 + 2n + 1 + 2n = 4n + 4$
  - $O(n)$

Worst Case Running Time – Example 2

```
beAnnoying(List n){
    List m = [];
    for (i=0; i < n.size(); i++){
        m.add(n[i]);
        for (j=0; j< n.size(); j++){
            print (“Hi, I’m annoying”);
        }
    }
}
```

Questions to ask:

- What are the units of the input size?
- What are the operations we’re counting?
- For each line:
  - How many times will it run?
  - How long does it take to run?
  - Does this change with the input size?

Worst Case Running Time – Example 2

```
beAnnoying(List n){
    List m = [];
    for (i=0; i < n.size(); i++){ // n times
        m.add(n[i]);
        for (j=0; j< n.size(); j++){ // n times
            print ("Hi, I'm annoying"); // 1
        }
    }
}
```

Questions to ask:

- What are the units of the input size?
  - # items
- What are the operations we're counting?
  - Adding or printing
  - Printing: $O(n^2)$
- For each line:
  - How many times will it run?
  - How long does it take to run?
  - Does this change with the input size?

Amortized Analysis Analogy

- Suppose I'd like to park in a lot where they charge $10 per day to park
- If you are caught in the lot without paying you are given a warning
- If you get 3 warnings, you are charged a $25 fine, and your warnings reset.
- Should you actually pay to park?
  - If you pay every day then you pay an average of $10 per day
  - If you do not pay then for every three days parking costs $0+$0+$25, for an average of $8.33 per day
    - Worst case analysis: parking costs $25
    - Amortized analysis: parking costs $8.33

Amortized Complexity Example - ArrayList

```
public void add(T value){
    if(data.length == size)
        resize();
    data[size] = value;
    size++;
}
private void resize(){
    T[] oldData = data;
    data = (T[]) new Object[data.length*2];
    for(int i = 0; i < oldData.length; i++)
        data[i] = oldData[i];
}
```

- What is the worst case running time of add?
  - Input size: size of “this”
  - Operations counted: indexing
  - $O(n)$

Amortized Complexity Example - ArrayList

```
public void add(T value){
    if(data.length == size)
        resize();
    data[size] = value;
    size++;
}
private void resize(){
    T[] oldData = data;
    data = (T[]) new Object[data.length*2];
    for(int i = 0; i < oldData.length; i++)
        data[i] = oldData[i];
}
```

Every time we resize, we earn `data.length` more adds guaranteed to not resize!

- Amortized Analysis Idea:
  - Suppose we have a program that in total does n adds.
  - How much time was spent "on average" across all n?
- Let c be the initial size of `data`
  - The first c adds take: $c + c = 2c$
  - The next $2c$ adds: $2c + 2c = 4c$
  - The next $4c$ adds: $4c + 4c = 8c$
  - Overall: c adds take an average of $2c$ time

Searching in a Sorted List

5	8	13	42	75	79	88	90	95	99
0	1	2	3	4	5	6	7	8	9

```java
public static boolean contains(List<Integer> a, int k){
    for(int i=0; i< a.size(); i++){
        if (a.get(i) == k)
            return true;
    }
    return false;
}
```

Faster way?

5	8	13	42	75	79	88	90	95	99
0	1	2	3	4	5	6	7	8	9

Can you think of a faster algorithm to solve this problem?

Binary Search

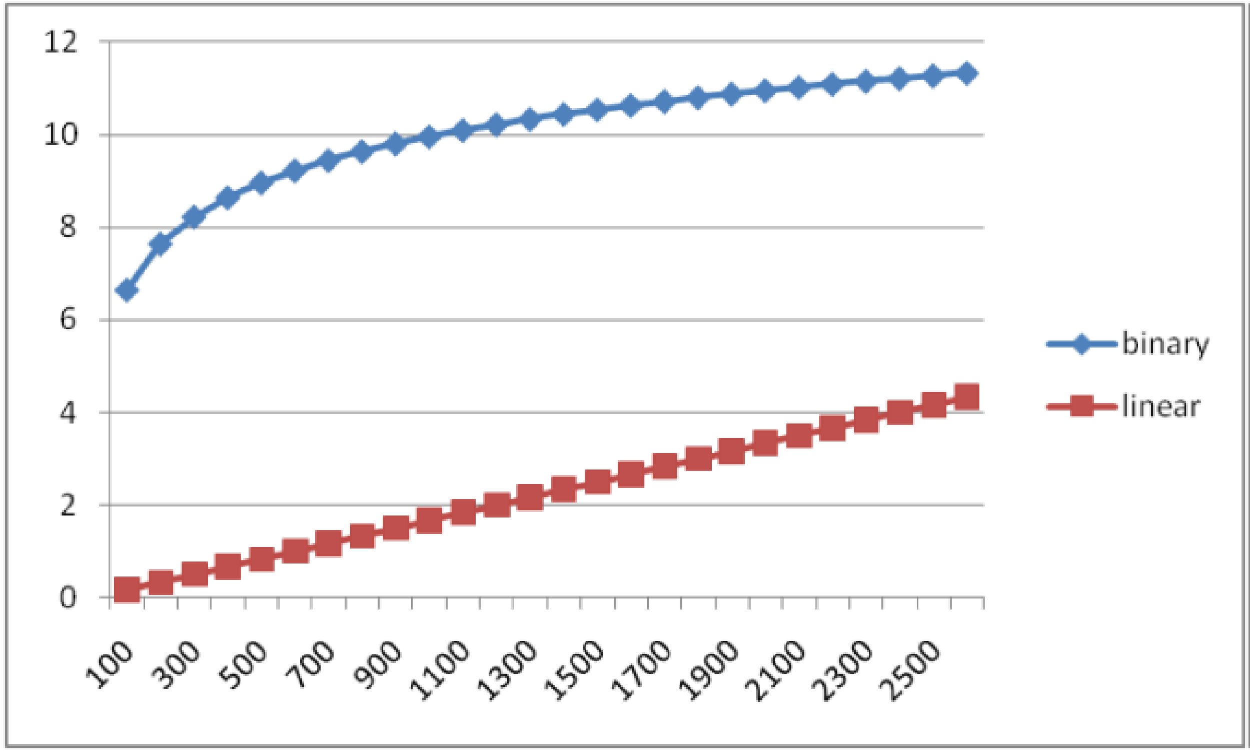
5	8	13	42	75	79	88	90	95	99
0	1	2	3	4	5	6	7	8	9

```java
public static boolean contains(List<Integer> a, int k){
    int start = 0;
    int end = a.size();
    while(start < end){
        int mid = start + (end-start)/2;
        if(a.get(mid) == k)
            return true;
        else if(a.get(mid) < k)
            start = mid+1;
        else
            end = mid;
    }
    return false;
}
```

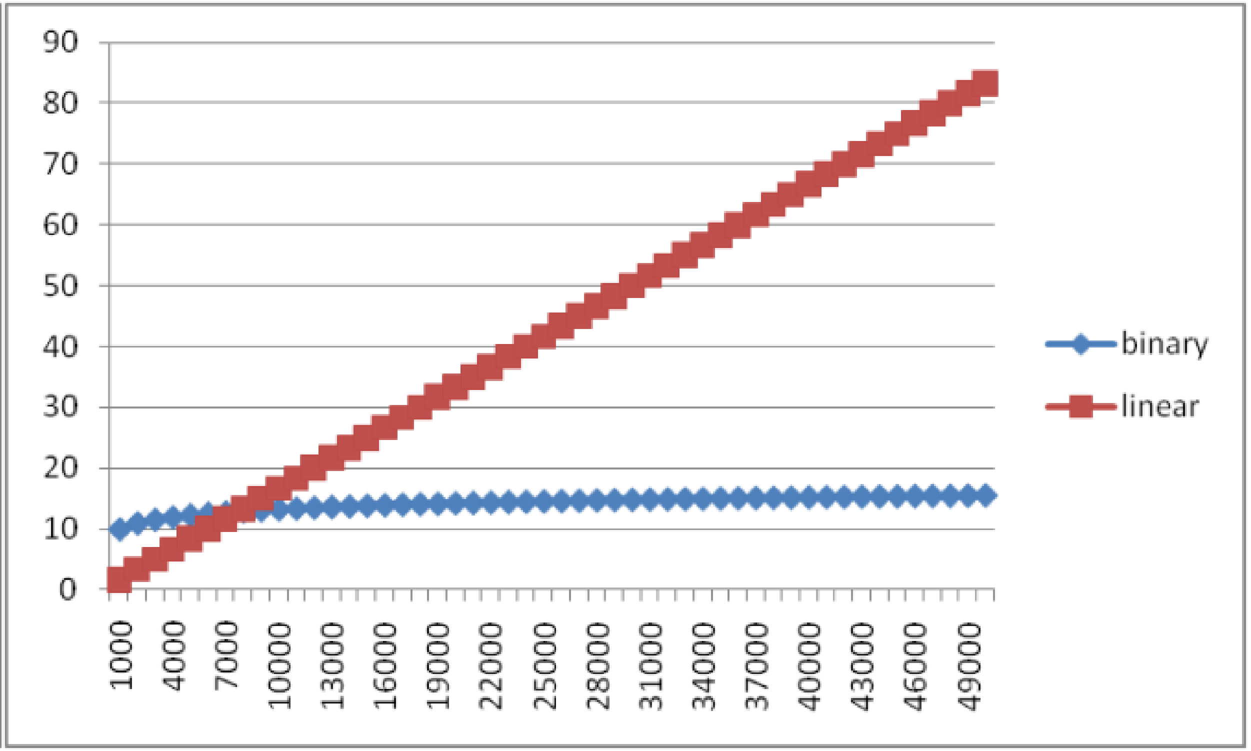
Why is this $\log_2 n$?

- In the beginning: `end-start`=n
- After 1 iteration: `end-start`=$\frac{n}{2}$
  - `mid-start = (start+(end-start)/2)-start`
  - `end-mid = end-(start+(end-start)/2)`
- Each iteration cuts the “gap” in half!
- We stop when the gap is 1

Comparing



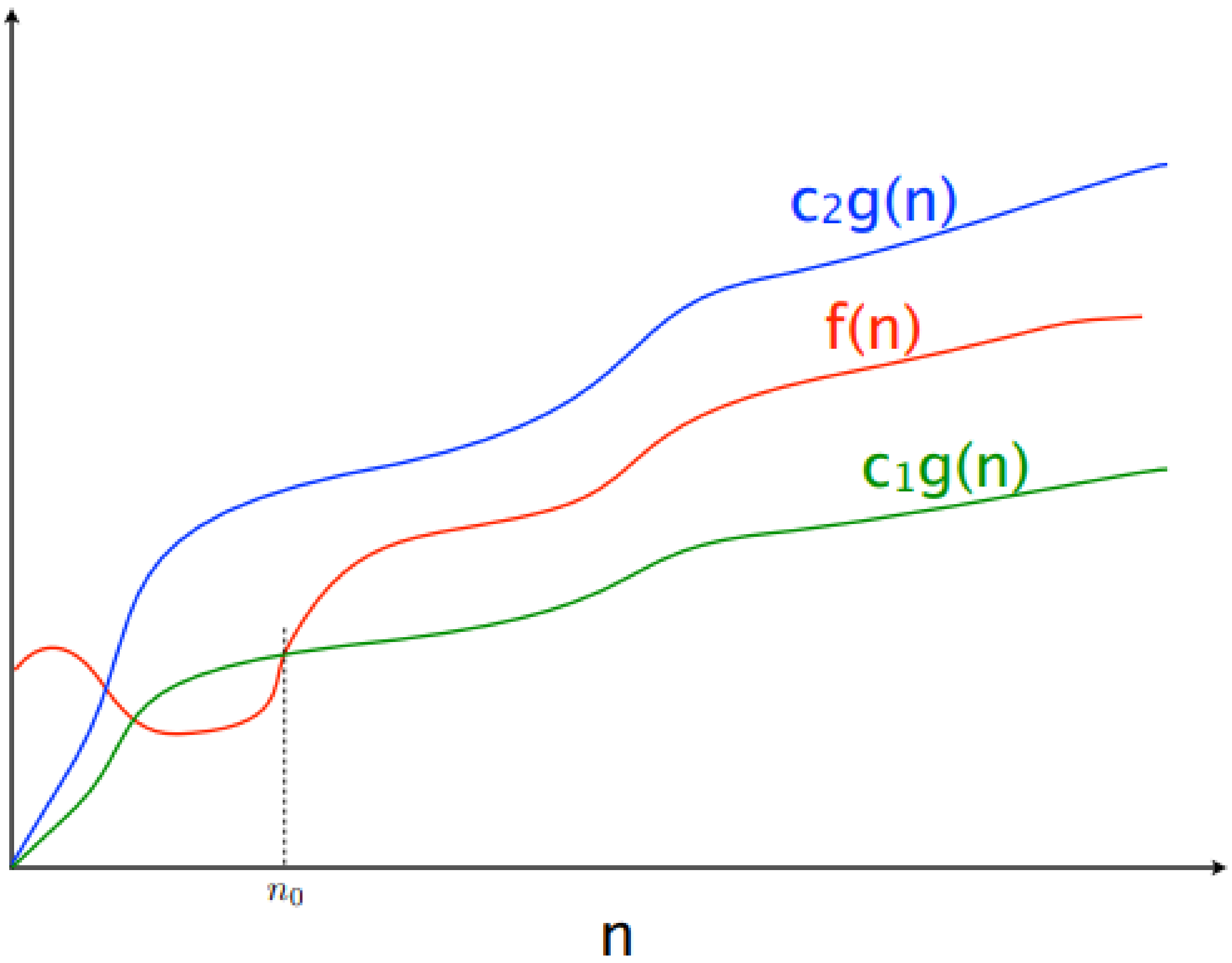
12
10
8
6
4
2
0
100
300
500
700
900
1100
1300
1500
1700
1900
2100
2300
2500
binary
linear



90
80
70
60
50
40
30
20
10
0
1000
4000
7000
10000
13000
16000
19000
22000
25000
28000
31000
34000
37000
40000
43000
46000
49000
binary
linear

Comparing Running Times

- Suppose I have these algorithms, all of which have the same input/output behavior:
  - Algorithm A's worst case running time is $10n + 900$
  - Algorithm B's worst case running time is $100n - 50$
  - Algorithm C's worst case running time is $\frac{n^2}{100}$
- Which algorithm is best?



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

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

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

Asymptotic Notation

- $O(g(n))$
  - The **set of functions** with asymptotic behavior less than or equal to $g(n)$
  - Upper-bounded by a constant times g for large enough values n
  - $f \in O(g(n)) \equiv \exists c > 0. \exists n_0 > 0. \forall n \geq n_0. f(n) \leq c \cdot g(n)$
- $\Omega(g(n))$
  - the **set of functions** with asymptotic behavior greater than or equal to $g(n)$
  - Lower-bounded by a constant times g for large enough values n
  - $f \in \Omega(g(n)) \equiv \exists c > 0. \exists n_0 > 0. \forall n \geq n_0. f(n) \geq c \cdot g(n)$
- $\Theta(g(n))$
  - "Tightly" within constant of g for large n
  - $\Omega(g(n)) \cap O(g(n))$

Idea of Θ

- $x = y$
  - $x \leq y \land x \geq y$

Asymptotic Notation Example

- Show: $10n + 100 \in O(n^2)$
  - **Technique:** find values $c > 0$ and $n_0 > 0$ such that $\forall n > n_0.\ 10n + 100 \leq c \cdot n^2$
  - **Proof:**

Asymptotic Notation Example

- Show: $10n + 100 \in O(n^2)$
  - **Technique:** find values $c > 0$ and $n_0 > 0$ such that $\forall n \geq n_0 .\ 10n + 100 \leq c \cdot n^2$
  - **Proof:** Let $c = 10$ and $n_0 = 6$. Show $\forall n \geq 6 .\ 10n + 100 \leq 10n^2$

$$10n + 100 \leq 10n^2$$
$$\equiv n + 10 \leq n^2$$
$$\equiv 10 \leq n^2 - n$$
$$\equiv 10 \leq n(n-1)$$

This is True because $n(n-1)$ is strictly increasing and $6(6-1) > 10$

Asymptotic Notation Example

- Show: $13\mathrm{n}^2 - 50\mathrm{n} \in \Omega(n^2)$
  - **Technique:** find values $c > 0$ and $n_0 > 0$ such that $\forall n \geq n_0.\, 13n^2 - 50n \geq c \cdot n^2$
  - **Proof:**
  - $c =$
  - $n_0 =$

Asymptotic Notation Example

- Show: $13\mathrm{n}^2 - 50\mathrm{n} \in \Omega(n^2)$
  - **Technique:** find values $c > 0$ and $n_0 > 0$ such that $\forall n \geq n_0.\, 13n^2 - 50n \geq c \cdot n^2$
  - **Proof:** let $c = 12$ and $n_0 = 50$. Show $\forall n \geq 50.\, 13n^2 - 50n \geq 12n^2$

    $13n^2 - 50n \geq 12n^2$

    $\equiv n^2 - 50n \geq 0$

    $\equiv n^2 \geq 50n$

    $\equiv n \geq 50$

    This is certainly true $\forall n \geq 50$.

Asymptotic Notation Example

- Show: $n^2 \notin O(n)$
- Want to show that there does not exist a pair of c and n_0 such that $\forall n_0 > n. n^2 \leq c \cdot n$

Asymptotic Notation Example

Proof by Contradiction!

- To Show: $n^2 \notin O(n)$
  - **Technique: Contradiction**
  - **Proof:** Assume $n^2 \in O(n)$. Then $\exists c, n_0 > 0$ s.t. $\forall n > n_0, n^2 \leq cn$
    Let us derive constant c. For all $n > n_0 > 0$, we know:
    $cn \geq n^2$,
    $c \geq n$.

    Since c is lower bounded by n, c cannot be a constant and make this True.
    Contradiction. Therefore $n^2 \notin O(n)$.

Gaining Intuition

- When doing asymptotic analysis of functions:
  - If multiple expressions are added together, ignore all but the “biggest”
    - If $f(n)$ grows asymptotically faster than $g(n)$, then $f(n) + g(n) \in \Theta\big(f(n)\big)$
  - Ignore all multiplicative constants
    - $f(n) + c \in \Theta\big(f(n)\big)$ for any constant $c \in \mathbb{R}$
  - Ignore bases of logarithms
  - Do NOT ignore:
    - Non-multiplicative and non-additive constants (e.g. in exponents, bases of exponents)
    - Logarithms themselves
- Examples:
  - $4n + 5$
  - $0.5n\log n + 2n + 7$
  - $n^3 + 2^n + 3n$
  - $n\log(10n^2)$

More Examples

- Is each of the following True or False?
  - $4 + 3n \in O(n)$
  - $n + 2\log n \in O(\log n)$
  - $\log n + 2 \in O(1)$
  - $n^{50} \in O(1.1^n)$
  - $3^n \in \Theta(2^n)$

Common Categories

- $O(1)$ "constant"
- $O(\log n)$ "logarithmic"
- $O(n)$ "linear"
- $O(n \log n)$ "log-linear"
- $O(n^2)$ "quadratic"
- $O(n^3)$ "cubic"
- $O(n^k)$ "polynomial"
- $O(k^n)$ "exponential"

Defining your running time function

- Worst-case complexity:
  - max number of steps algorithm takes on “most challenging” input
- Best-case complexity:
  - min number of steps algorithm takes on “easiest” input
- Average/expected complexity:
  - avg number of steps algorithm takes on random inputs (context-dependent)
- Amortized complexity:
  - max total number of steps algorithm takes on M “most challenging” consecutive inputs, divided by M (i.e., divide the max total sum by M).

Beware!

- Worst case, Best case, amortized are ways to select a function
- O, Ω, Θ are ways to compare functions
- You can mix and match!
- The following statements totally make sense!
  - The worst case running time of my algorithm is $\Omega(n^3)$
  - The best case running time of my algorithm is $O(n)$
  - The best case running time of my algorithm is $\Theta(2^n)$