

CSE 332 Autumn 2026

Lecture 2: Algorithm Analysis

pt.1

Ruth Anderson

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

Announcements

- Preliminary Survey
 - Due Monday 10/05
 - On Canvas
- Exercise 0 released
 - Due Friday 10/09/2026
 - There are 2 separate Gradescope submissions
- “Meet the staff” activity
 - Sometime during the first 4 weeks of class, visit a CSE 332 office hour (in person or on zoom)
 - Tell the staff member you want to get checked off
 - You do not have to have a question about course content
 - We just want to meet you!
- Exercise 1 Running Time: coming early next week, due Mon Oct 12

What do we care about in an algorithm?

- Correctness:
 - Does the algorithm do what is intended.
- Performance:
 - Speed time complexity
 - Memory space complexity
- Why analyze?
 - To make good design decisions
 - Enable you to look at an algorithm (or code) and identify the bottlenecks, etc.

Pick an algorithm with the best running time

How should we compare two algorithms?

- Uh, why NOT just run the program and time it??
 - Too much *variability*, not reliable or *portable*:
 - Hardware: processor(s), memory, etc.
 - OS, Java version, libraries, drivers
 - Other programs running
 - Implementation dependent
 - Choice of input
 - Testing (inexhaustive) may *miss* worst-case input
 - Timing does not *explain* relative timing among inputs (what happens when n doubles in size)
- Often want to evaluate an *algorithm*, not an implementation
 - Even *before* creating the implementation (“coding it up”)

Wish list for algorithm comparison method:

So we are not going to run the program and time it...what should we do?

Ideally we want a method to compare two algorithms that is:

- *Independent* of CPU speed, programming language, coding tricks, etc.
- Tells us how algorithms compare on *large input sizes* (Input size = n)
 - E.g. any correct sorting algorithm is fast enough if you are only sorting 10 things
- We can use without implementing the algorithm and running it

We will do “Resource Analysis”

- Allows us to compare *algorithms*, not implementations
 - Using observations necessarily couples the algorithm with its implementation
 - If my implementation on my computer takes more time than your implementation on your computer, we cannot conclude your algorithm is better
- We can predict an algorithm’s running time before implementing
- Understand where the bottlenecks are in our algorithm

Process for Resource Analysis

- End Result: A *function* which maps the algorithm's input size to **count of resources used**
 - Input of the function: **sizes** of the input
 - Number of characters in a string, number of items in a list, number of pixels in an image
 - Output of the function: **counts** of resources used
 - Typically we think of this as “amount of time” but we will specifically count things like: number of times the algorithm adds two numbers together, number times the algorithm does a > or < comparison, maximum number of bytes of memory the algorithm uses at any time
- Important note: Make sure you know the “units” of your input and output!

Running Time Analysis

- Units of “time”
 - Operations
 - Whichever operations we pick
- How do we express running time?
 - Function
 - Domain (input): size of the input
 - (e.g., number of elements in our array, number of characters in our string)
 - Range: count of operations

Scenarios to analyze for an algorithm

- 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).

Usually: 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

Analysis Process From CSE 123/143

- Count the number of “primitive operations”
 - +, -, compare, arr[i], arr.length, etc
- Write that count as an expression using n (the input size)
- Put that expression into a “bucket” by
 - ignoring constants and
 - Ignoring “non-dominant” terms,
 - then put a $O()$ around it.
- Examples:
 - $4n^2 + 8n - 10$ ends up as $O(n^2)$
 - $\frac{1}{2}n + 80$ ends up as $O(n)$
 - $n(n + 1)$ ends up as $O(n^2)$

Analysis Process For Us

- Pick a Scenario/Case to analyze (usually worst)
- Count the number of *chosen* operation(s)
 - Factors to consider when choosing which operation(s) to count
 - **Necessity**: should be necessary for solving the problem
 - **Frequency**: should be the most frequently done (up to a constant factor)
 - **Magnitude**: should be expensive to perform each chosen operation
- Write that count as an expression using n (the input size)
- Put an asymptotic bound on it (one of O , Ω , Θ)
 - More on this next class

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 – Example 1 (Questions)

```
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 – Example 1 (Answers)

```
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 – Example 2 (Questions)

```
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 – Example 2 (Answers)

```
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?

Reminder: Scenarios for 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 (distribution-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).

ArrayList - Worst Case Time of Add

```
public void add(int value){
    if(data.length == size)
        resize();
    data[size] = value;
    size++;
}

private void resize(){
    int[] oldData = data;
    data = new int[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
 - $O(n)$

ArrayList – Amortized Time of Add

```
public void add(int value){
    if(data.length == size)
        resize();
    data[size] = value;
    size++;
}

private void resize(){
    int[] oldData = data;
    data = new int[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 before the next 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: $\frac{\text{total time}}{\# \text{ of adds}} = \frac{14c}{7c} = 2$

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
 - This is an amortized analysis

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

```
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

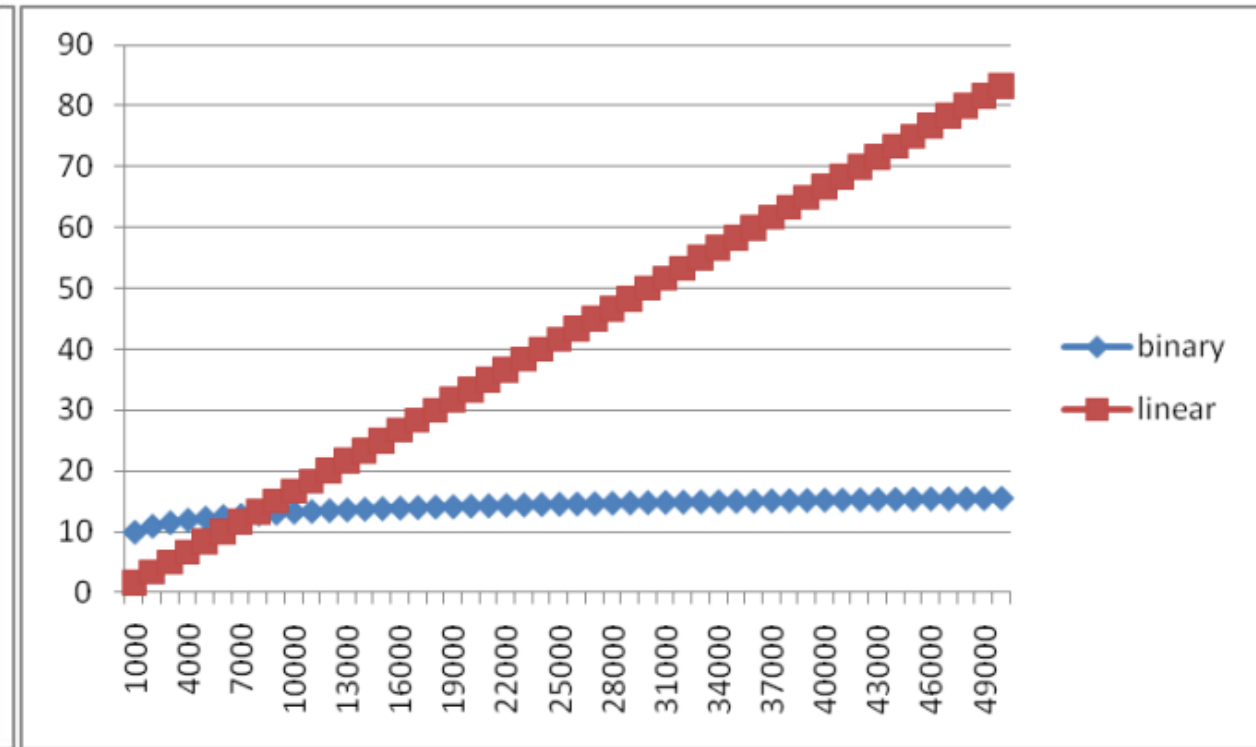
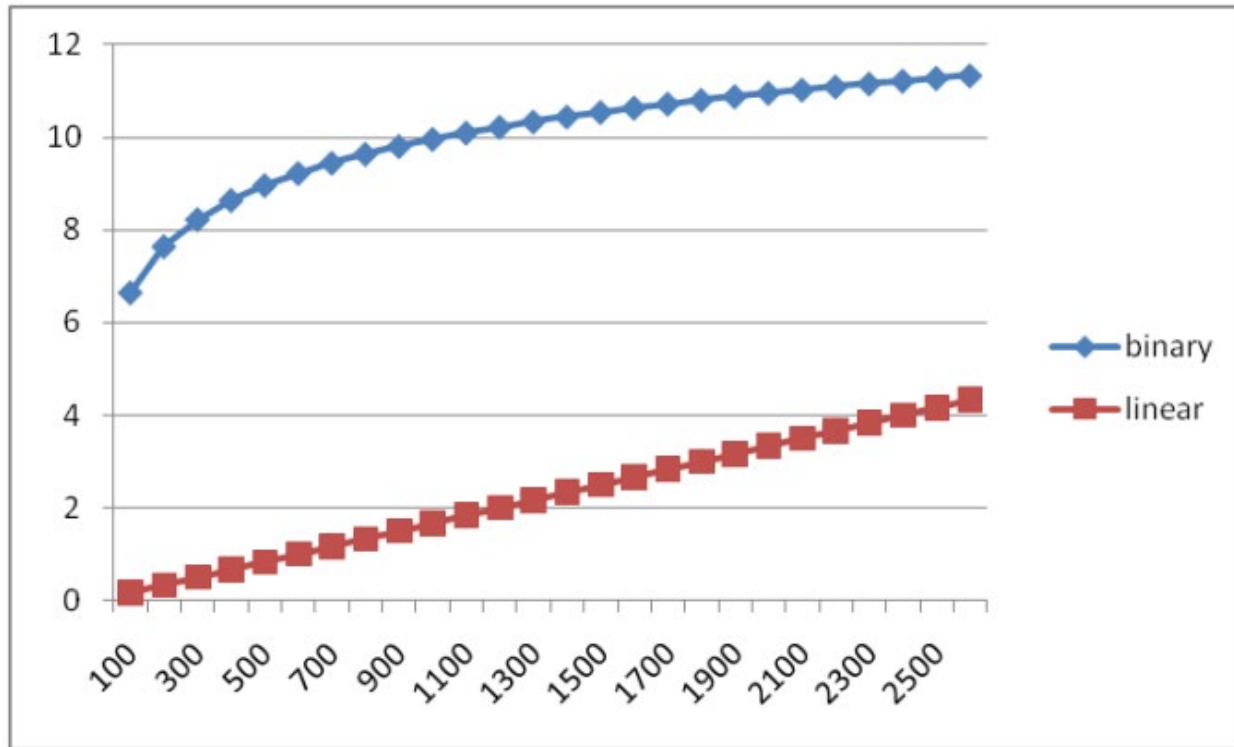
5	8	13	42	75	79	88	90	95	99
0	1	2	3	4	5	6	7	8	9

```
public static boolean binary_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: $\text{end} - \text{start} = n$
- After 1 iteration: $\text{end} - \text{start} = \frac{n}{2}$
 - $\text{mid} - \text{start} = (\text{start} + (\text{end} - \text{start}) / 2) - \text{start}$
 - $\text{end} - \text{mid} = \text{end} - (\text{start} + (\text{end} - \text{start}) / 2)$
- Each iteration cuts the “gap” in half!
 - x iterations the size is $\frac{n}{2^x}$
- We stop when the gap is 1
 - Solve for x : $\frac{n}{2^x} = 1$

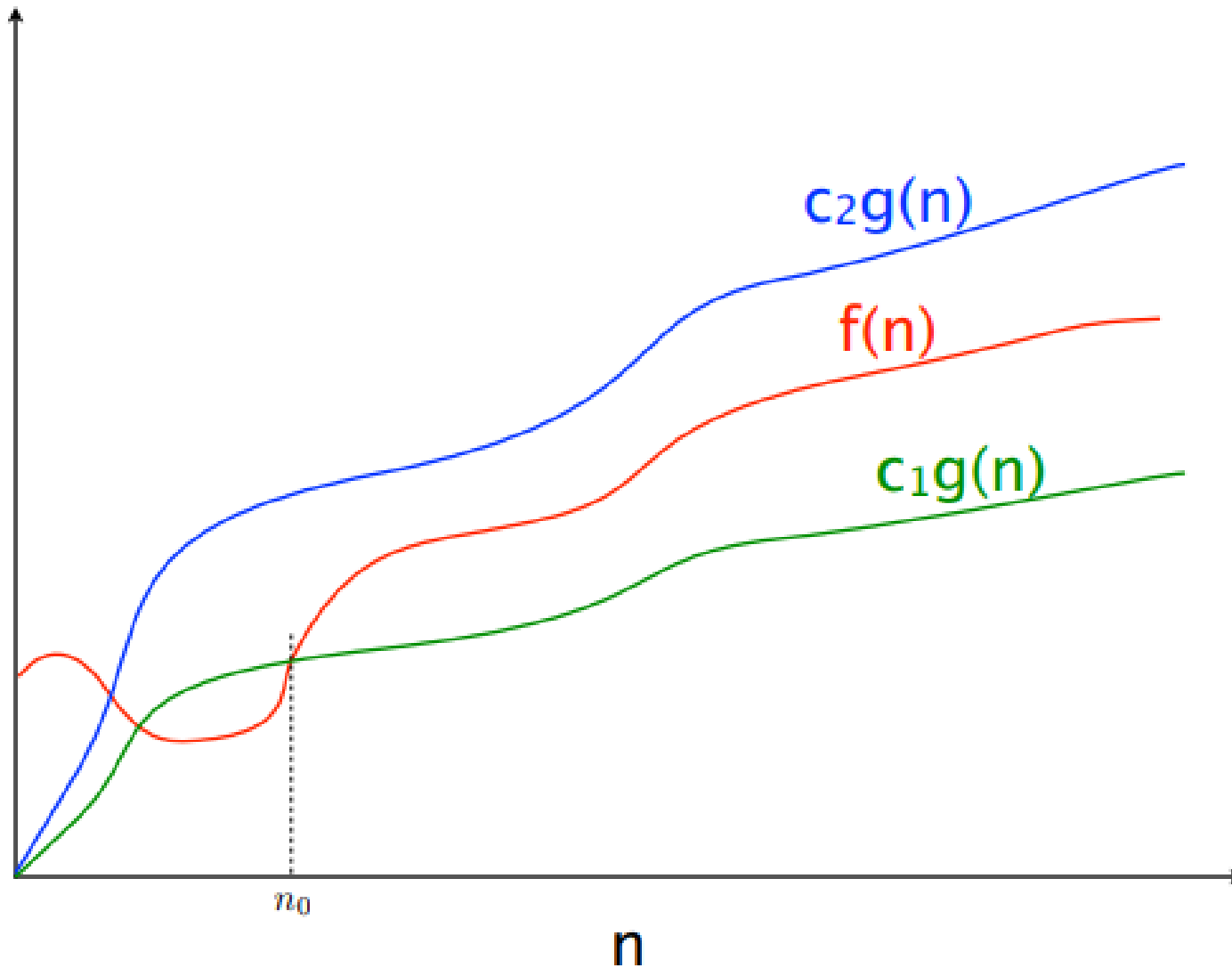
Running Time Graphed



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?

Asymptotically comparing $f(n)$ with $g(n)$



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

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

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

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

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