CSE 332
Data Structures & Parallelism

Algorithm Analysis

Melissa Winstanley
Spring 2024

Today & Next Time - Algorithm Analysis

- What do we care about?
- How to compare two algorithms
- Analyzing code
- Asymptotic Analysis
- Big-Oh Definition

What do we care about?

- 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.

Q: How should we compare two algorithms? Sort all students who have taken 332

Chandni

5 seconds

3.5 seconds

0 seconds

Arya

4 seconds

2 seconds

Today - Algorithm Analysis

- What do we care about?
- How to compare two algorithms
- Analyzing code
- Asymptotic Analysis
- Big-Oh Definition

A: 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")

Comparing algorithms

When is one algorithm (not implementation) better than another?

- Various possible answers (clarity, security, …)
- But a big one is performance: for sufficiently large inputs, runs in less time (our focus) or less space

Large inputs (n) because probably any algorithm is "plenty good" for small inputs (if n is 10, probably anything is fast enough)

Answer will be independent of CPU speed, programming language, coding tricks, etc.

Answer is general and rigorous, complementary to "coding it up and timing it on some test cases"

- Can do analysis before coding!

Goals for Algorithm Analysis

- Identify a *function* which maps the algorithm's input size to a measure of resources used
  - Input of the function: **size** of the function input (**n**)
    - 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
- Important note: Make sure you know the "units" of your domain and codomain!

Today - Algorithm Analysis

- What do we care about?
- How to compare two algorithms
- Analyzing code
  - **How to count different code constructs**
  - Best Case vs Worst Case
  - Ignoring Constant Factors
- Asymptotic Analysis
- Big-Oh Definition

Analyzing code

Basic operations take "some amount of" constant time

- Arithmetic
- Assignment
- Access one Java field or array index
- Etc.

$O(1)$

(This is an *approximation of reality*: a very useful "lie".)

Consecutive statements	Sum of time of each statement
Loops	Num iterations * time for loop body
Conditionals	Time of condition plus time of slower branch
Function Calls	Time of function's body
Recursion	Solve recurrence equation

Example 1

```
b = b + 5
c = b / a
b = c + 100
```

```
for (i = 0; i < n; i++) {
    sum++;
}
```

```
if (j < 5) {
    sum++;
} else {
    for (i = 0; i < n; i++) {
        sum++;
    }
}
```

Example 2

```
int coolFunction(int n, int sum) {
    int i, j;
    for (i = 0; i < n; i++) {
        for (j = 0; j < n; j++) {
            sum++;
        }
    }
    print "This program is great!"
    for (i = n; i > 1; i = i / 2) {
        sum++;
    }
    return sum
}
```

Using Summation for Loops

```
for (i = 0; i < n; i++) {
    sum++;
}
```

$$\sum_{i=0}^{n-1} 5 = 5n$$

Example 3

```
List<Integer> beAnnoying(List<Integer> lst){
    List m = new ArrayList<Integer>();
    for (i = 0; i < lst.size(); i++){
        m.add(lst.get(i));
        for (j = 0; j < lst.size(); j++){
            print("Hi, I'm annoying");
        }
    }
    return m;
}
```

What about memory?

```
List<Integer> beAnnoying(List<Integer> lst){
    List m = new ArrayList<Integer>();
    for (i = 0; i < lst.size(); i++){
        m.add(lst.get(i));
        for (j = 0; j < lst.size(); j++){
            print("Hi, I'm annoying");
        }
    }
    return m;
}
```

$5n^2 + 3n$

$O(n)$

$$\sum_{0}^{n-1}\left(\sum_{0}^{n-1} 5 + 3\right)$$

Today - Algorithm Analysis

- What do we care about?
- How to compare two algorithms
- Analyzing code
  - How to count different code constructs
  - **Best Case vs Worst Case**
  - Ignoring Constant Factors
- Asymptotic Analysis
- Big-Oh Definition

Complexity cases

We'll start by focusing on two scenarios:

- Worst-case complexity: max # steps algorithm takes on "most challenging" input of size N

- Best-case complexity: min # steps algorithm takes on "easiest" input of size N

Example - best case? worst case?

```
b = b + 5
c = b / a
b = c + 100
```

constant

```
for (i = 0; i < n; i++) {
    sum++;
}
```

3

```
if (j < 5) {
    sum++;
} else {
    for (i = 0; i < n; i++) {
        sum++;
    }
}
```

Example

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

Find an integer in a *sorted array*

```
// requires array is sorted
// returns whether k is in array
boolean find(int[]arr, int k){
    ???
}
```

Linear search - Best Case & Worst Case

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

```
// requires array is sorted
// returns whether k is in array
boolean find(int[]arr, int k){
  for(int i=0; i < arr.length; ++i)
    if(arr[i] == k)
      return true;
    return false;
  }
}
```

Best case:

Worst case:

Linear search - Best Case & Worst Case

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

```
// requires array is sorted
// returns whether k is in array
boolean find(int[]arr, int k){
  for(int i=0; i < arr.length; ++i)
    if(arr[i] == k)
      return true;
    return false;
  }
}
```

Best case: 6-ish steps = O(1)

Worst case:
  5-ish steps * (arr.length) = O(n)

Remember a faster search algorithm?

binary logn

linear n

Worst case analysis

- Worst-case complexity: max # steps algorithm takes on "most challenging" input of size N
  - Does NOT depend on how big N is
  - Depends on the actual arguments to the algorithm

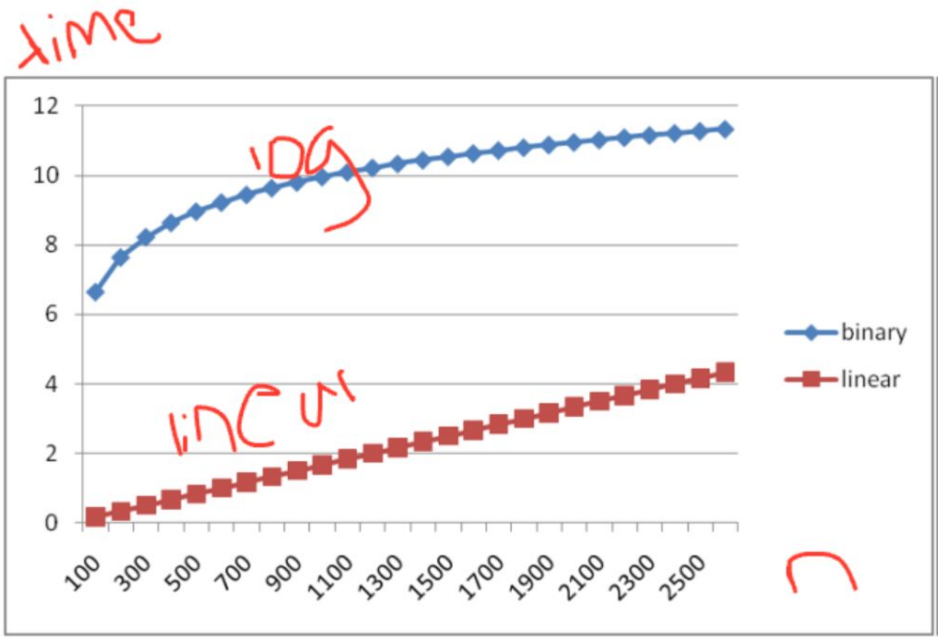
Today - Algorithm Analysis

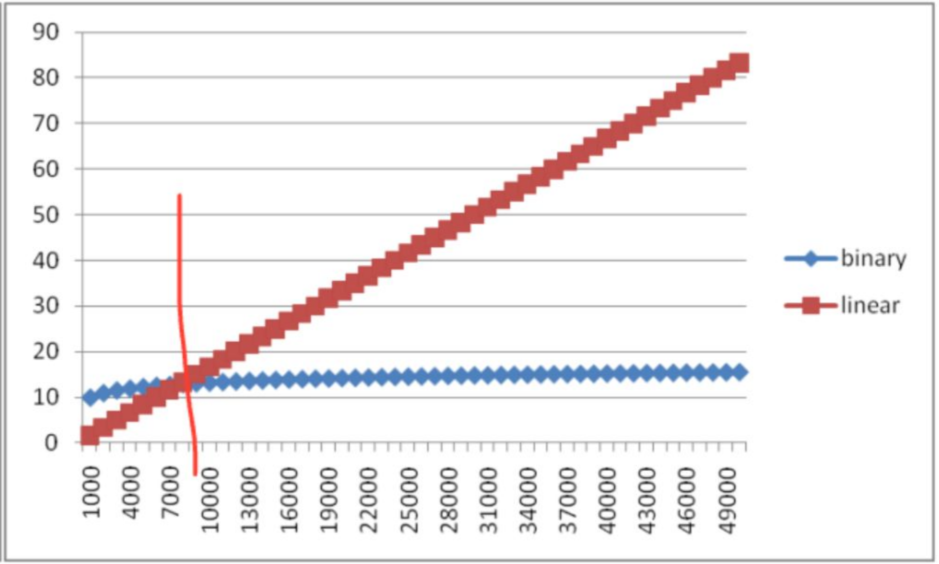
- What do we care about?
- How to compare two algorithms
- Analyzing code
  - How to count different code constructs
  - Best Case vs Worst Case
  - **Ignoring Constant Factors**
- Asymptotic Analysis
- Big-Oh Definition

Ignoring constant factors

- So binary search is O(log n) and linear is O(n)
  - But which will actually be faster?
  - Depending on **constant factors** and **size of n**; in a particular situation, linear search could be faster….
- Could depend on constant factors
  - How *many* assignments, additions, etc. for each n
- And could depend on size of n – what if n is small?
- **But** there exists some n_0 such that for all $n > n_0$ binary search "wins"
- Let's look at a couple plots to get some intuition…

Linear search vs Binary search





Let's give linear search a boost (n / 600)