Lecture 5
Algorithm Performance Analysis

Department of Computer Science
Hofstra University

Lecture Goals

- Calculate the big-O class of complicated code snippets.
- Define worst case, average case, and best case performance and describe why each of these is used.
- State and justify the asymptotic performance for linear search, binary search, selection sort, insertion sort, merge sort, and quick sort.
- Recognize and avoid some common pitfalls in asymptotic analysis.
- Use Java timing libraries to measure execution time.
- Use runtimes from a real system to reason about performance.
- Identify components of real systems which impact execution time.

Motivation

Algorithm: a strategy for solving a problem.

Performance: how good that strategy is.

Algorithm with good performance can answer very hard questions in very short amount of time. We need to have a sense of how good our algorithm is without just running it.

There is hereby imposed on the taxable income of every individual (other than a surviving spouse as defined in section 2(a) or the head of a hou as defined in section 2(b)) who is not a married individual (as defined in section termined in accordance with the following table:

How long dose this take?

If you are single, never se, and not the head of a household, you pay taxes according to the followi e:

Use flesch score to measure of text readability

$$\text{FleschScore} = 206.835 - 1.015\left(\frac{\#\text{words}}{\#\text{sentences}}\right) - 84.6\left(\frac{\#\text{syllables}}{\#\text{words}}\right)$$

Problem with just looking at the "stopwatch" time.

- different computers
- different compilers
- different libraries/optimizations

The time for running the specific code on a specific machine on a specific input

Is NOT a good representation of how good our algorithm is.

Performance Analysis Overview

- Count number of operations instead of time’
  - # operations is independent of the hardware platform
- Focus on how performance scales with large input size
  - Asymptotic Performance Analysis

Algorithm runtime

```
boolean hasLetter(String word, char letter)
{
        for (int i = 0; i < word.length(); i++) {
                if (word.charAt(i) == letter) {
                        return true;
                }
        }
        return false;
}
```

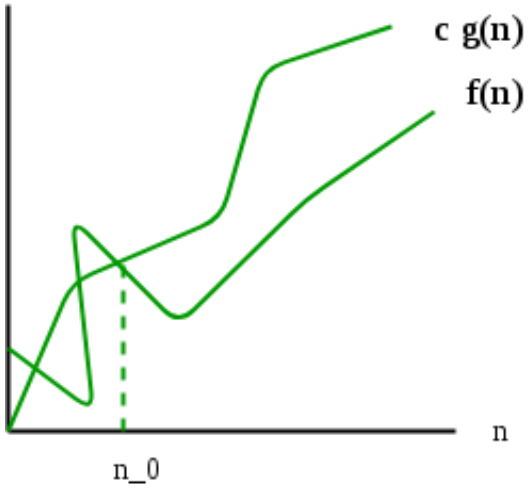
- Algorithm runtime is dependent on the inputs.
- hasLetter(“Happy”, a): search for the letter “a” in the word “Happy”, loop runs for 2 iterations before finishing and returning true.
- hasLetter(“Happy”, x): search for the letter “x” in the word “Happy”, loop runs for 5 iterations before finishing and returning false.

Asymptotic Analysis

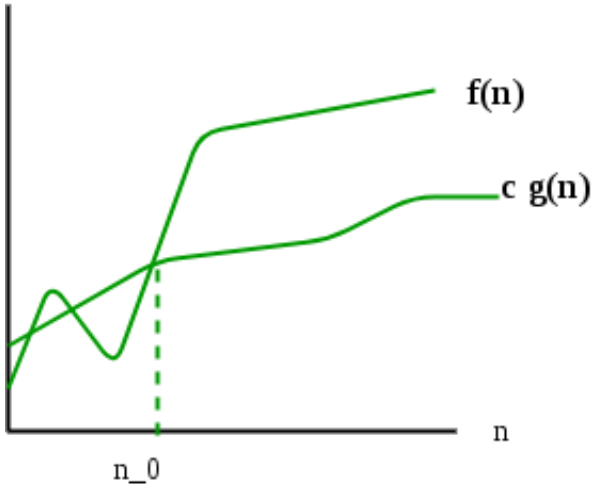
- Asymptotic analysis examines how functions behave as their input grows arbitrarily large. It focuses on the limiting behavior of functions for large input size, rather than their exact values for specific inputs.
  - Runtime as input size n gets large, as we don't care if the algorithm runs for 10 ms vs. 2 s with small input size n; we care if it runs for 100 s vs. 100 hours/days/years for very large n.
- Big-O Notation (O-notation) denotes upper bound
- Theta Notation (Θ-notation) denotes exact bound
- Omega Notation (Ω-notation) denotes exact bound

Three Notations

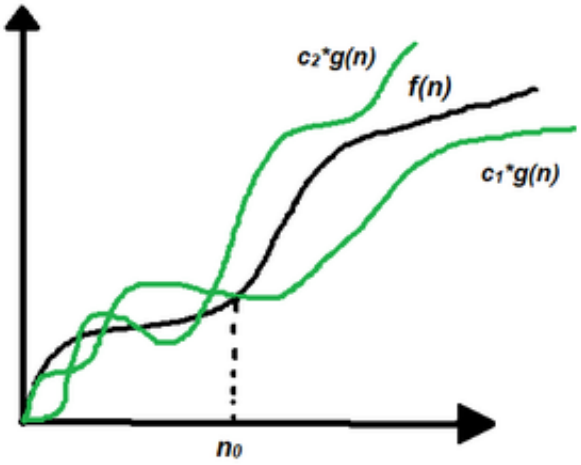
- Let f(n) and g(n) be two functions that map from the set of natural numbers to the set of natural numbers.
- Big O denotes upper bound: Function f(n) is said to be O(g(n)) if there exist a positive constant c and n_0 such that, $0 \leq f(n) \leq cg(n)$ for all $n \geq n_0$
- Big Omega Ω denotes lower bound: Function f is said to be Ω(g(n)), if there exist a positive constant c and n_0 such that, $0 \leq cg(n) \leq f(n)$ for all $n \geq n_0$
- Big Theta Θ denotes exact bound: Function f is said to be Θ(g), if there exist constants c1, c2 > 0 and a natural number n_0 such that $c1*g(n) \leq f(n) \leq c2*g(n)$ for all $n \geq n_0$



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



f(n) = Omega(g(n))

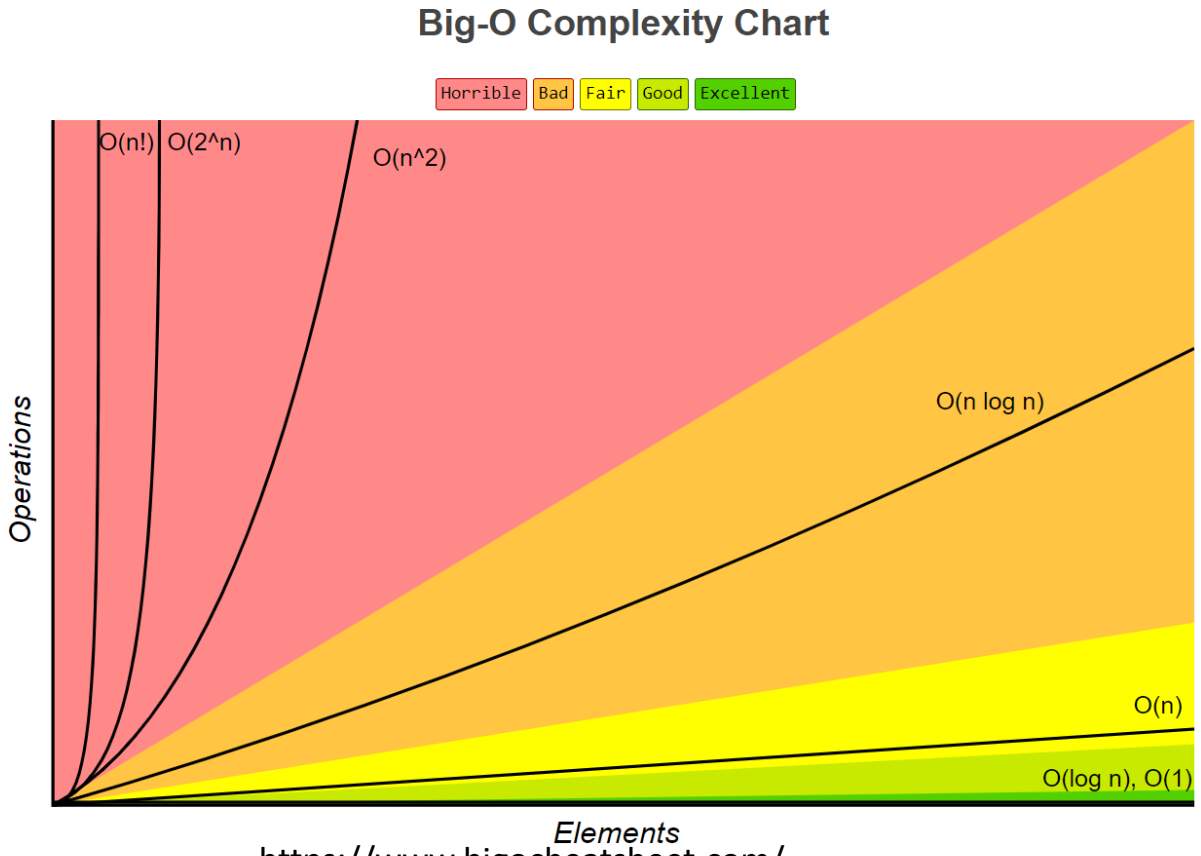


f(n) = Θ(n)

https://www.geeksforgeeks.org/types-of-asymptotic-notations-in-complexity-analysis-of-algorithms/

Big-O Complexity Chart

- We use big-O notation to express how algorithm performance scale with input size.
  - Two functions are in the same big-O class if they have the same rate of growth with increasing input size.
  - $f_1(n)$ dominates $f_2(n)$ if $O(f_2(n)) < O(f_1(n))$, i.e., $f_1(n)$ has higher asymptotic complexity than $f_2(n)$ in big O notation
- O(1) < O(logn) < O(n) < O(nlogn) < O(n²) < O(2") < O(n!)



https://www.bigocheatsheet.com/

Big-O Notation Examples

- Rate of growth determined by the dominating highest-order term; ignore lower-order terms, constant terms, and coefficients in big-O notation.
- Examples:
  - $f(n) = C_1 n + C_2$: complexity O(n)
  - $f(n) = C_1 \log n + C_2$: complexity O(log n)
  - $f(n) = C_3 n + C_4 \log\log n + C_5$: complexity O(n)
  - $f(n) =$ 100, 1000000, log2000, 10^4: complexity O(1)
  - $f(n) =$ n/4, 2n+3, n/100 + log n, n + 10000, log n + 10: complexity O(n)
  - $f(n) =$ n^2 + n, $2n^2$, n^2 + 1000n, n^2 + n log n + n, n^2/10000: complexity O(n^2)
- Also works for multiple variables:
  - $f(n, m) =$ 100n^2 + 1000m + n. Asymptotic complexity O(n^2 + m)
  - $f(n, m) =$ 1000m^2 + 200mn + 30m + 20n. Asymptotic complexity O(m^2 + mn)

Big-O notation in 5 minutes
https://www.youtube.com/watch?v= vX2sjlpXU

Big-O Notation in 3 Minutes
https://www.youtube.com/watch?v=x2CRZaN2xgM

Exercise

- 1. Suppose algorithm running time for input size n is $g(n) = 2^n + n^2 + 100$, what is its complexity in big-O notation?
- ANS: $O(2^n)$
- For $g(n) = 3n \log n + 4 \log n \ + n^2 + n$,
- ANS: $O(n^2)$
- For $g(n) = 3n \log n + 4 \log n \ + n$,
- ANS: $O(n \log n)$
- 2. If Algorithm 1 has complexity $O(\log n)$, Algorithm 2 has complexity $O(n^2)$, will Algorithm 1 always have fewer operations (shorter running time) than Algorithm 2?
- ANS: No. If Algorithm 1 has running time $100000 * \log n$, Algorithm 2 has running time $3n^2$, then $100000 * \log n > 3n^2$ for small n.

Exercise Con't

- 3. Suppose algorithm running time for input size n is
  - $n^2 + n + \log n$
  - $n * (n - i) + n + \log n$ (i is a loop iteration variable within 1 to n)
  - $0.001 * n^2 + 1000 * n + 10000 * \log n$
  - $(n + 100)^2 + (100 * (n + 100000)) + 100 * \log n$

  What is the big O notation for the algorithm complexity in each case?
- ANS: $O(n^2)$
- What is the answer if 2^n is added to each term?
- ANS: $O(2^n)$

Loop Complexity Analysis Examples

```
for (int i = 0; i < n; i = i+c) {
  //Some O(1) code
}
```

i = 0, c, 2c, … < n, number of iterations $\left\lceil \frac{n}{c} \right\rceil$

Complexity O(n)

```
for (int i = n; i > 0; i = i-c) {
  //Some O(1) code
}
```

i = n, n-c, n-2c, … > n, number of iterations $\left\lceil \frac{n}{c} \right\rceil$

Complexity O(n)

```
for (int i = 1; i < n; i = i*c) {
  //Some O(1) code
}
```

$i = c^0, c^1, c^2, \ldots, c^{k-1} < n$, number of iterations $k < \log_c n+1$

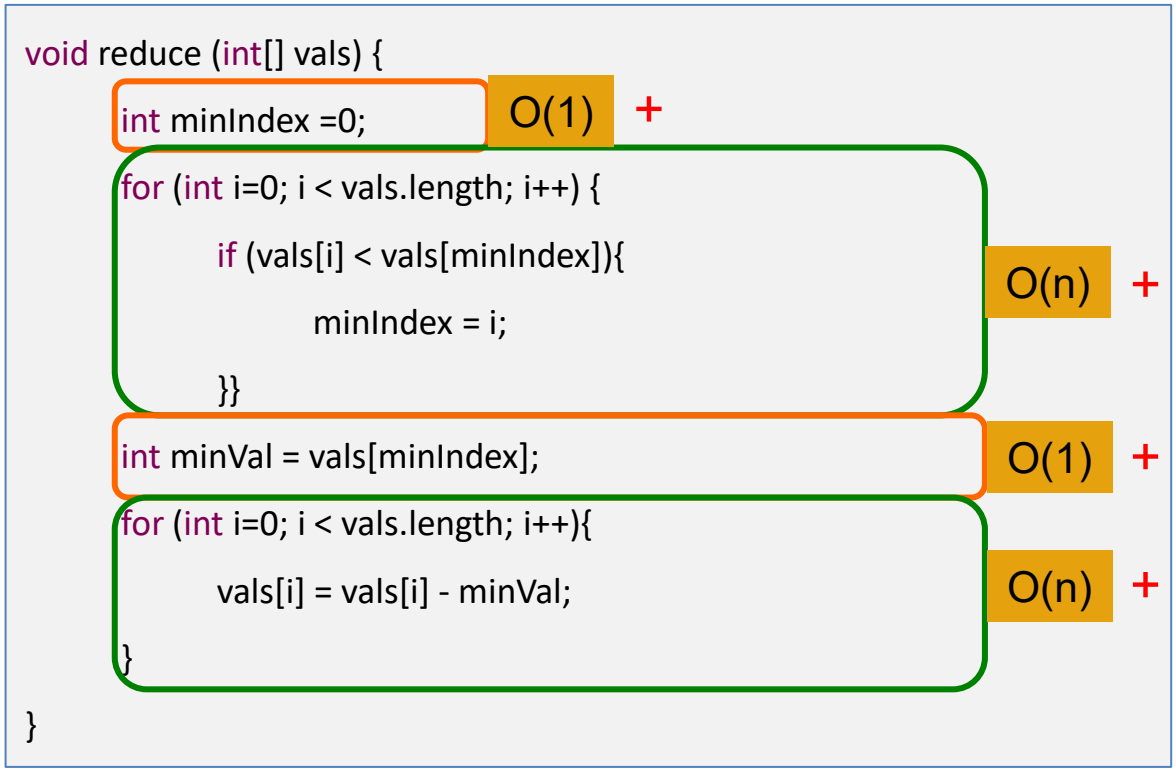
Complexity $O(\log_c n)$

```
for (int i = n; i > 1; i = i/c) {
  //Some O(1) code
}
```

$i = n/c^0, n/c^1, n/c^2, \ldots, n/c^{k-1} > 1$, number of iterations $k < \log_c n+1$

Complexity $O(\log_c n)$

Consecutive Loops



- The first for loop finds the minimum value of the array vals with size n. It runs for n iterations, each taking constant time O(1), hence it has linear complexity O(n).
- The second for loop reduces each value in the array by the minimum value. It runs for n iterations, each taking constant time O(1), hence it has linear complexity O(n).
  - e.g., vals = [1,2,5,3] before, [0,1,4,2] after.
- Two assignment statements each has complexity O(1)
- Total complexity of the entire algorithm is linear O(n) (the result of O(n)+O(n)+O(1)+O(1)).

Nested Loops

```
int maxDifference (int[] vals) {
    int max = 0;                                   O(1)
    for (int i=0; i < vals.length; i++) {
        for (int j=0; j < vals.length; j++) {
            if (vals[i] – vals[j] > max) {
                max = vals[i] – vals[j];
            }
        }                                          O(n)
    }                                              O(n^2)
    return max;                                    O(1)
}
```

- The nested for loops look for the maximum difference between any two array elements.
  - e.g., vals = [1,7,2,4,6,8], return 8 – 1 = 7
- Inner loop runs for n iterations, each taking constant time O(1), hence it has linear complexity O(n).
- Outer loop runs for n iterations, each taking linear time O(n), hence it has quadratic complexity $O(n^2)$.

Exercises

```
int fun (int n) {
   for (int i = 0; i < n; i = i+c) {
     //Some O(1) code
   }
    for (int i = 0; i < n; i = i*2) {
     //Some O(1) code
   }
  for (int i = 0; i < 100; i = i++) {
     //Some O(1) code
   }
}
```

- First loop has complexity O(n)
- Second loop has complexity O(log n)
- Third loop has complexity O(1)
- Total complexity is O(n), result of O(n)+O(log n)+O(1)

```
int fun (int n) {
   for (int i = 0; i < n; i++) {
      for (int j = 1; j < n; j = j*2) {
        //Some O(1) code
      }
   }
}
```

- Inner loop has complexity O(log n)
- Outer loop has complexity O(n)
- Total complexity is O(n log n), result of O(n)*O(log n)

Exercises Con't

```
int fun (int n) {
   for (int i = 0; i < n; i++) {
      for (int j = 1; i < n; j = j*2) {
        //Some O(1) code}
   }
  for (int i = 0; i < n; i++) {
      for (int j = 1; j < n; j++) {
        //Some O(1) code}
   }
}
```

- First nested loop has complexity O(n log n)
- Second nested loop has complexity $O(n^2)$
- Total complexity is $O(n^2)$, result of O(n log n) + $O(n^2)$

```
int fun (int n) {
   for (int i = 0; i < n; i++) {
      for (int j = 1; i < n; j = j*2) {
        //Some O(1) code}
   }
  for (int i = 0; i < m; i++) {
      for (int j = 1; j < m; j++) {
        //Some O(1) code}
   }
}
```

- First nested loop has complexity O(n log n)
- Second nested loop has complexity $O(m^2)$
- Total complexity is O(n log n + m^2), result of O(n log n) + $O(m^2)$

Best-case, Average-case, Worst-case Complexity

- Best-case complexity: best possible performance of algorithm for any input of size n
- Worst-case complexity: worst possible performance of algorithm for any input of size n
- Average-case complexity: performance on average, consider all possible inputs of size n
  - Often mathematically difficult to analyze
- In big-O notation, Best-case complexity $\leq$ Average-case complexity $\leq$ Worst-case complexity

Best-case, Average-case, Worst-case Complexity Example 1

```
boolean hasLetter(String word, char letter)
{
        for (int i = 0; i < word.length(); i++) {
                if (word.charAt(i) == letter) {
                        return true;
                }
        }
        return false;
}
```

- Algorithm complexity w.r.t input size n = word.length(), for Best-case, Average-case, and Worst-case.
- Best-case: O(1), if word starts with letter
  - hasLetter("apple", "a");
- Worst-case: O(n), if letter at the end (or missing)
  - hasLetter("happy", "x");
  - hasLetter("happy", ”y");
- Average-case: O(n), assuming letter may take on any random value

Best-case, Average-case, Worst-case Complexity Example 2

```
int getSum (int arr[], int n)
{
        if (n%2 == 0) {
                return 0; }
        int sum = 0;
        for (int i=0; i<n; i++) {
                sum += arr[i]; }
        return sum;
}
```

- Algorithm complexity w.r.t input n = size of input array arr[], for Best-case, Average-case, and Worst-case, for a made-up function that returns 0 for even n, and sum of array elements for odd n. (So the name getSum() is not accurate.)
- Best-case: O(1), if n is even (n%2 == 0)
- Worst-case: O(n), if n is odd (n%2 != 0)
- Average-case: O(n), assuming n may be even or odd with equal chance

Analyzing Search Algorithms

	Best Case	Worst Case
Linear Search	O(1)	O(n)
Binary Search*	O(1)	O(log(n))

* Assuming data is sorted

sorting cost?

times to half size?

How many times can we divide by 2 before we get to 1?

Linear Search: Basic Algorithm

Start at the first **index** in the array

while **index < length** of the array:
    if toFind matches current value,
        return true
    increment index by 1

return false

E.g. hasLetter(String word, **char** letter)

Binary Search: Basic Algorithm

Initialize **low = 0, high = length of list**

while low <= high:
    mid = (high+low)/2
    if toFind matches value at mid,
        return true
    if toFind < value at mid
        high = mid-1 first half
    else low = mid+1 second half
return false

Worst case: don't find!

Cuts search base in half at each iteration, so the total # iterations is $\log_2(n)$

Binary search in 4 minutes
https://www.youtube.com/watch?v=fDKIpRe8GW4

$\log_{10}(n) = O(\log_2(n))$ since: $\log_{10}(n) = \frac{\log_2(n)}{\log_2(10)}$

Additional Resources

- Big-O analysis
  - http://web.mit.edu/16.070/www/lecture/big_o.pdf -- Big O handout from MIT
  - https://www.interviewcake.com/article/java/big-o-notation-time-and-space-complexity -- explanation of Big O with examples
  - http://discrete.gr/complexity/ -- "A Gentle Introduction to Algorithm Complexity Analysis" GIves a lot more detail than what we provided.
- Sorting algorithms
  - http://www.java2novice.com/java-sorting-algorithms/ -- 5 different sort algorithm explanation with codes
  - https://www.cs.cmu.edu/~adamchik/15-121/lectures/Sorting%20Algorithms/sorting.html -- different search algrotihms with solid examples
- Timing code in Java
  - http://stackoverflow.com/questions/180158/how-do-i-time-a-methods-execution-in-java -- many ways offered by many people