Honors Data Structures

Lecture 4: Introduction to asymptotic notation

1/31/22

Daniel Bauer

Algorithm Analysis

- An algorithm is a clearly specified set of simple instructions to be followed to solve a problem.
- Algorithm Analysis — Questions:
    - Does the algorithm terminate?
    - Does the algorithm solve the problem? (correctness)
    - What resources does the algorithm use?
        - Time / Space

Analyzing Runtime

- We often want to compare several algorithms.
    - Compare between different algorithms how the runtime $T(n)$ grows with increasing input sizes n.
- We are using Java or Scala, but the same algorithms could be implemented in any language on any machine.
- How many basic operations/*steps* does the algorithm take? All operations assumed to have the same time.

Worst and Average case

- Usually the runtime depends on the type of input (e.g. sorting is easy if the input is already sorted).

- $T_{worst}(n)$: *worst case* runtime for the algorithm on ANY input. The algorithm is **at least** this fast.

- $T_{average}(n)$: *Average case analysis* — expected runtime on typical input.

- $T_{best}(n)$: Rarely, we are interested in the *best case analysis*.

Asymptotic Notation, big-Θ

- How does the running time *T(n)* increase as the input size n increases, *in the limit?*

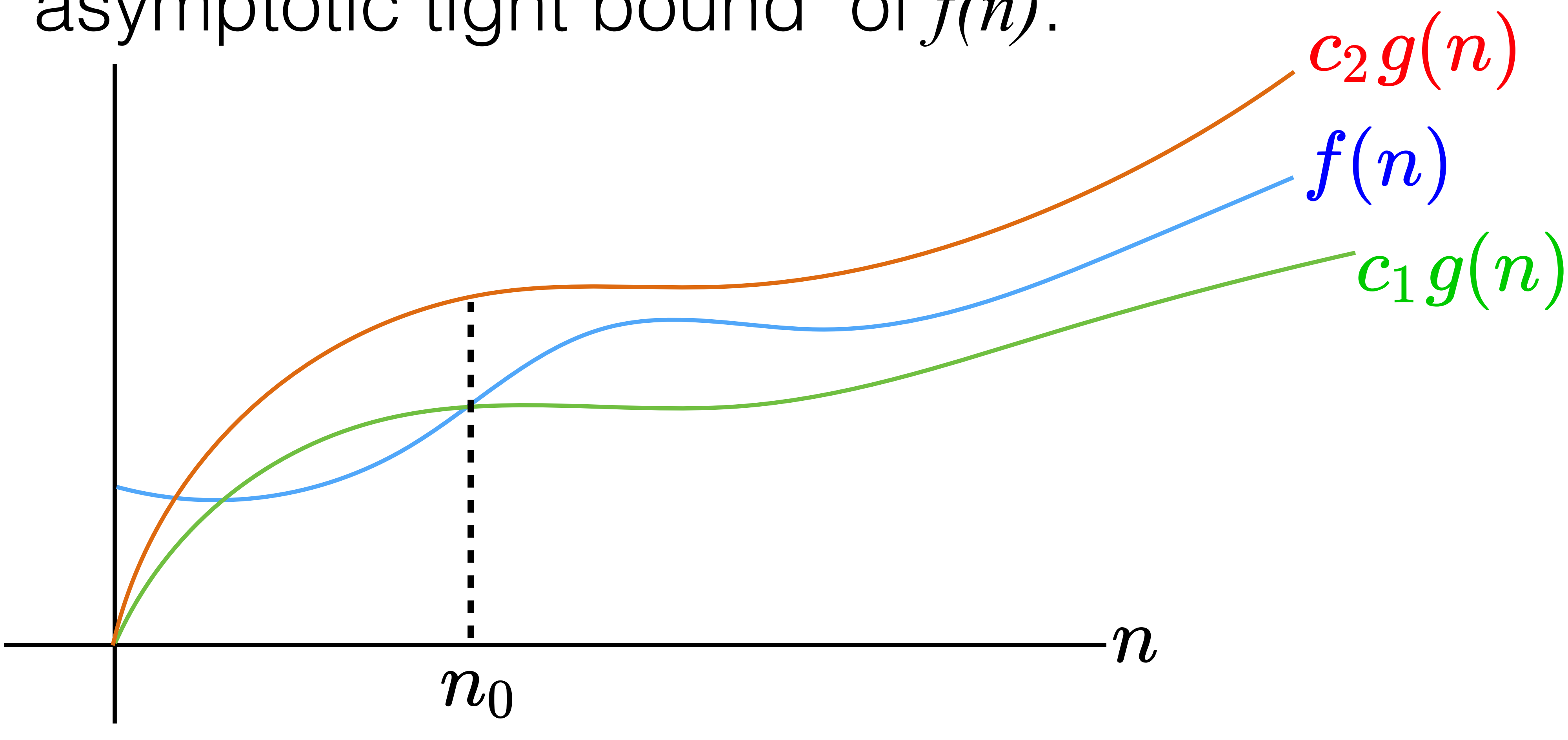
- $\Theta(g(n))$ is the set of functions with the same *growth rate* as $g(n)$. Instead of $T(n) \in \Theta(g(n))$ we usually write $T(n) = \Theta(g(n))$.

- For example, the worst case running time of of binary search is in $\Theta(log_n)$.

Big-Θ - Definition

$$\Theta(g(n)) = \{f(n) : \text{there exist positive constants } c_1, c_2, \text{and } n_0 \text{ such that } 0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n) \text{ for all } n \geq n_0\}.$$

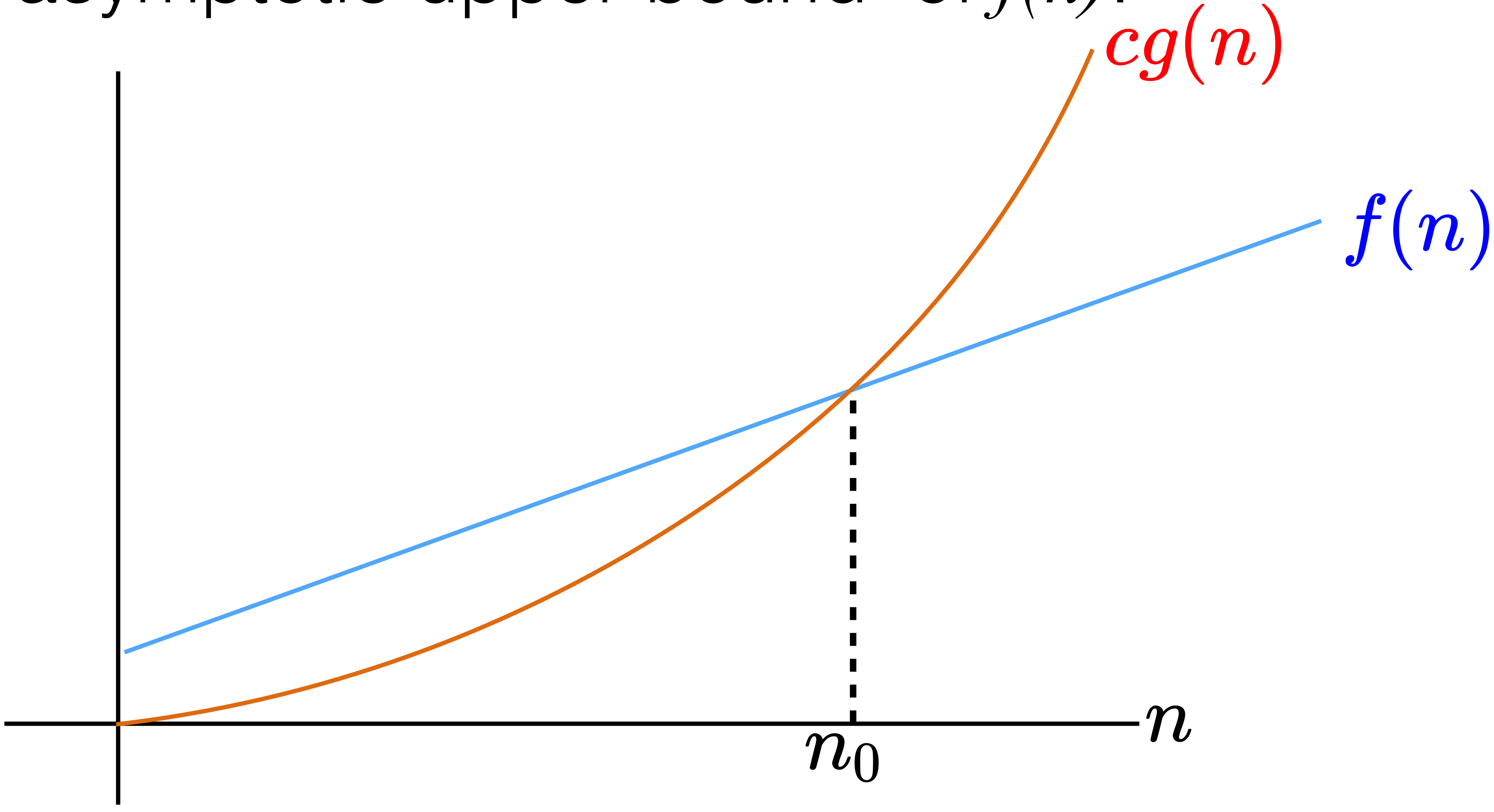
g(n) is an "asymptotic tight bound" of *f(n)*.



Comparing Function Growth: Big-O

$$\Omega(g(n)) = \{f(n) \text{ there exist positive constants } c \text{ and } n_0 \text{ such that } f(n) \le cg(n) \text{ for all } n \ge n_0\}.$$

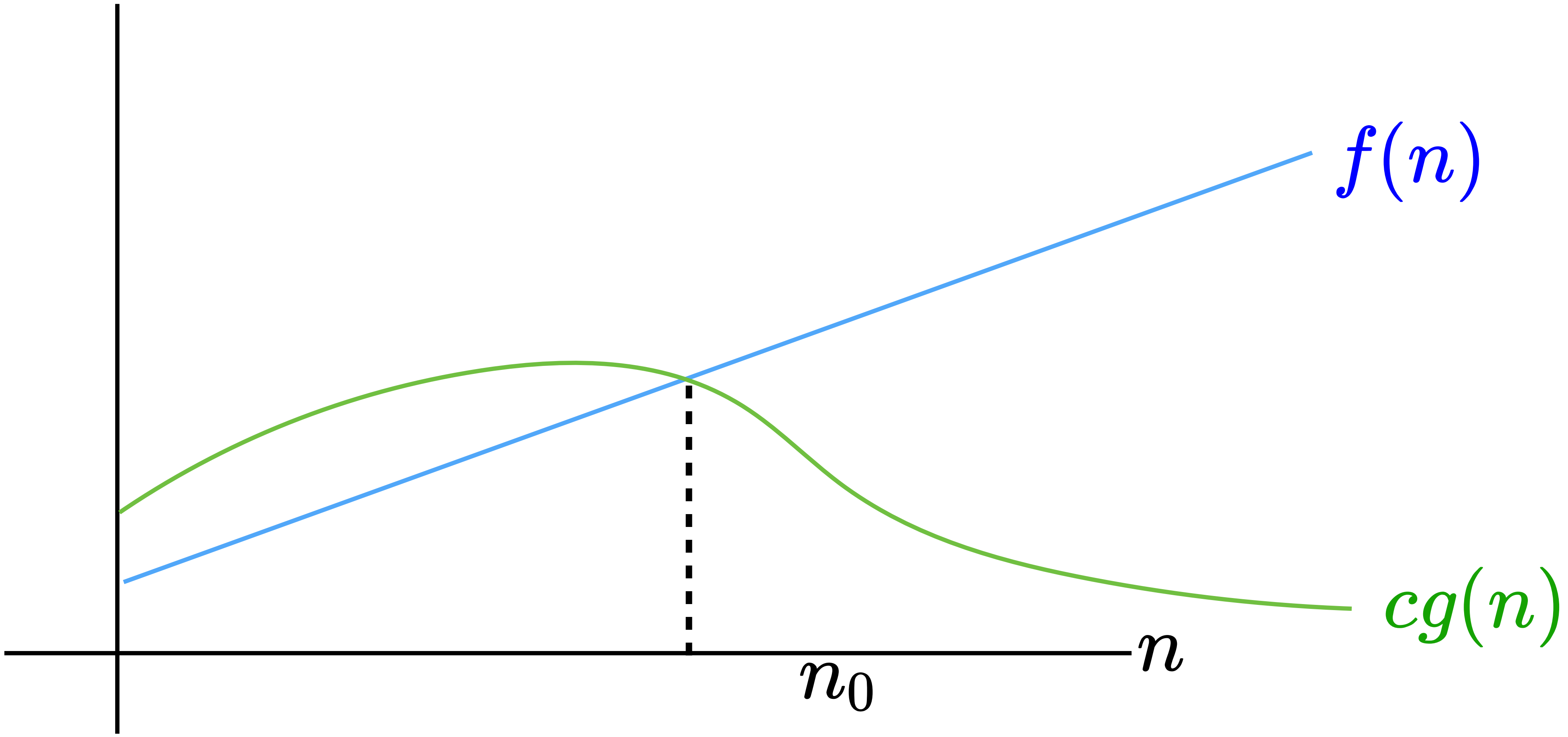
g(n) is an "asymptotic upper bound" of *f(n)*.



Comparing Function Growth: Big-Ω

$$\Omega(g(n)) = \{f(n) \text{ there exist positive constants } c \text{ and } n_0 \text{ such that } f(n) \geq cg(n) \text{ for all } n \geq n_0\}.$$

g(n) is an "asymptotic lower bound" of *f(n)*.



Rules for Big-O (1)

If $T(N)$ is a polynomial of degree k then

$$T(N) = \Theta(N^k)$$

For instance: $9N^3 + 12N^2 - 5 = \Theta(N^3)$

$log^k(N) = (log(N))^k = O(N)$ for any k .

$log_a(N) = \Theta(log_2(N))$ for any a .

Rules for Big-O (2)

If $T_1(N) = O(f(N))$ and $T_2(N) = O(g(N))$ then

1. $$\begin{aligned} T_1(N) + T_2(N) &= O(f(N) + g(N)) \\ &= O(\max(f(N), g(N))) \end{aligned}$$

2. $$T_1(N) \cdot T_2(N) = O(f(N) \cdot g(N))$$

General Rules: Basic *for*-loops

Compute $\sum_{i=1}^{N} i^3$

```
public static int sum(int n){
  int partialSum = 0;
 for (int i = 1; i <= n; i++)
      partialSum += i * i * i;
  return partialSum;
}
```

1 step

1 step (initialization)

+1 step for last test

N iterations

2 steps each

4 steps each

1 step

$$T(N) = 6N + 4 = O(N)$$

(running time of statements in the loop) X (iterations)

If loop runs a constant number of times: $O(c) = O(1)$

Generally, we do not need to count individual steps!

General Rules: Nested *L*oops

Analyze inside-out.

```
for (i=0; i < n; i++)
  for (j=0; j < n; j++)
    k++;
```

N iterations $N \cdot O(N) = O(N^2)$

N iterations $O(N)$

1 step each $O(c)$

General Rules: Consecutive Statements

```
for (i = 0; i < n; i++)
  a[i] = 0;
for (i=0; i < n; i++)
  for (j = 0; j < n; j++)
      a[i] += a[j] + i + j;
```

$O(N)$

$O(N^2)$

$$O(N) + O(N^2) = O(N^2)$$

General Rules:
if/else conditionals

```
if (condition)
    S1
else
    S2
```

$T_{\text{test}}(N)$

$T_{S_1}(N)$

$T_{S_2}(N)$

$$T(N) = O(\max(T_{S_1}(N), T_{S_2}(N)) + T_{\text{test}}(N))$$

General Rules: calling methods

```
for (int i=0; i < n; i++) {
    someMethod(a[i]);
}
```

N steps

$T_{someMethod}(M)$

$$T(N) = N \cdot T_{someMethod}(M)$$

Logarithms in the Running Time

```
public class BinarySearch {

        public int binarySearch(Integer[ ] a, Integer x ) {
                int low = 0, high = a.length - 1;

                while( low <= high ) {
                        int mid = ( low + high ) / 2;

                        if( a[ mid ] < x )
                                low = mid + 1;
                        else if( a[ mid ] > x )
                                high = mid - 1;
                        else
                                return mid;    // Found
                }
                return -1;
        }
}
```

$log_2(N)$

$O(1)$

Each iteration of the while loop cuts remaining partition in half.
There are $\log_2$(N) iterations. The total runtime is $log_2(N) \cdot O(1) = O(\log N)$.