Algorithm Analysis

CS 4104: Data and Algorithm Analysis

Yoseph Berhanu Alebachew

May 11, 2025

Virginia Tech

Table of contents

Algorithm Analysis

Algorithm Analysis: Definition

- Analysis of algorithms refers to the determination of the amount of **resources** necessary to execute them.
- Resource include things like **time** and **storage**
- Most algorithms are designed to work with inputs of arbitrary length
- Efficiency or running time of an algorithm is stated as a function of input size
- The function relates input size to the amount of resource consumed during execution
- The unit of resource might be
  - Number of steps for time complexity
  - Storage locations for space complexity
  - Number of bytes transferred for bandwidth complexity

Algorithm Analysis: Dimensions of Analysis

- We are often concerned about **computational time** as resource requirement
- Sometime we are also concerned about **memory**, **communication bandwidth**, and other costs are considered.
- In this course we will always assume the resource we are most concerned is the computational time
- When we analyze multiple viable candidates, we will be able to identify which are the most efficient once and which are not.

Model of Computation

- Time efficiency estimates depend on what we define to be a step.
- One might consider addition of two number as one step
- This assumption may not be warranted in certain contexts.
  - For example, if the numbers involved in a computation may be arbitrarily large, the time required by a single addition can no longer be assumed to be constant.
- To be able to compare algorithms based on efficiency we need to define a standard of some sort.
- Before we can analyze an algorithm we must have a model of the implementation technology that we will use.

Model of Computation: Cost Models

- Two cost models are generally used:
  - **uniform cost model**
    - Also called uniform-cost measurement
    - Assigns a constant cost to every machine operation, regardless of the size of the numbers involved
  - **logarithmic cost model**
    - Also called logarithmic-cost measurement
    - Assigns a cost to every machine operation proportional to the number of bits involved
- Models should also set standards on parameters such as number of processors and memory access pattern and time.

Model of Computation: RAM Model

- For this course we use we assume a generic one processor, **random access machine (RAM)** model of computation.
- In the RAM model instructions are executed once after another with no concurrent operations.
- Strictly speaking we should precisely define the instructions of the RAM model and their cost
- Doing so would be tedious and offer little insight into algorithm design and analysis.
- We make assumptions about the instructions and leave out listing the available instructions and their cost explicitly.

Model of Computation: RAM Model

- We should not abuse this model
  - E.g., By assuming instruction that would not realistically be available in real computers such as an instruction to sort.
- RAM model has operations for
  - *Arithmetic* (add, subtract, multiply, divide, remainder, floor, and ceiling)
  - *Data movement* (load, store, copy)
  - *Control* (conditional and unconditional branch, subroutine call and return)
  - Basic data types like **integer** and **floating point**

Model of Computation: Other Models

- Some real computers sometimes have more instructions than those defined here
  - For example x^y is a multiple instruction operation but when $x = 2$ it can be done with a single instruction which is shifting.
- In real computer there exists a multi-level memory hierarchy (cache, primary memory and virtual memory)
- We do not consider these in the RAM model and in this course
- Other models (more in the lecture note)
  - External Memory Model
  - Cache Oblivious Model

Efficiency

- Remember: Different algorithms can be devised to solve a single problem.
- Each of these algorithms will differ dramatically in their efficiency.
- **Efficiency** is the comparison of what is actually produced or performed with what can be achieved with the same consumption of resources (money, time, labor, etc.).
- This difference in efficiency is even much more significant than the hardware and software the solutions are implemented on.

Efficiency: Insertion vs Merge Sort

- Two well know sorting algorithms are insertion sort and merge sort.
- for an input size of N and two independent constants $C_1 and\ c_2$
  - Insertion sort takes $C_1 N^2$ time
  - Merge sort takes $C_2 N \log N$ time
- After a certain input size merge sort clearly outperforms insertions sort
- For example $C_1 = 10$ and $C_2 = 200$, $N \simeq 150$

Efficiency: Insertion vs Merge Sort

N	Insertion Sort	Merge Sort	Efficient Algorithm
2	40	400	Insertion
10	1000	6643.85619	Insertion
20	4000	17287.7124	Insertion
30	9000	29441.3436	Insertion
40	16000	42575.4248	Insertion
50	25000	56438.5619	Insertion
60	36000	70882.6871	Insertion
70	49000	85809.9622	Insertion
80	64000	101150.85	Insertion
90	81000	116853.356	Insertion
100	100000	132877.124	Insertion
120	144000	165765.374	Insertion
140	196000	199619.924	Insertion
150	225000	216864.561	Merge
160	256000	234301.699	Merge
170	289000	251919.292	Merge
180	324000	269706.711	Merge
190	361000	287654.513	Merge

Table 1: Running time for insertion sort and merge sort with $c_1 = 10$ *and* $c_2 = 200$

Efficiency: Insertion vs Merge Sort

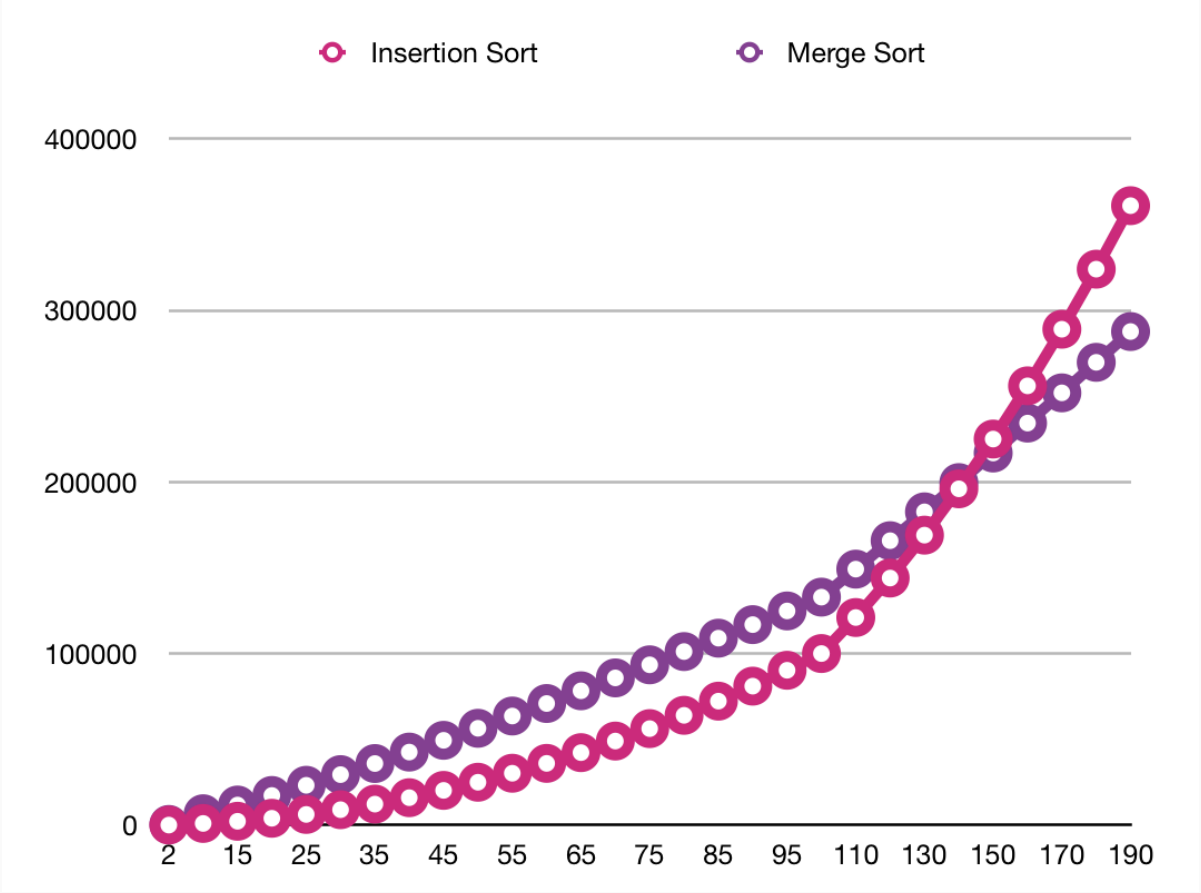


Figure 1: Asymptotic Comparison of running times of insertion sort and merge sort

Another Reason to Analyze Algorithms

- If speed and cost were not an issue do we still need to study how to analize to algorithms ?

Another Reason to Analyze Algorithms

- If speed and cost were not an issue do we still need to study how to analize to algorithms ?
  - Yes, We still have to study algorithms to show they actually terminate and do so with the right answer
  - This means proof their correctness.
  - In reality, though, both speed and cost associated with algorithms is an issue and we want to study it.

Example: Recap

- Analyzing algorithm refers to figuring out the amount of resource required to execute it.
- We do this in relation to its time complexity
  - Hence it refers to the time it takes to execute the algorithm
- Remember we assumed elementary operations in our computational model to take a a single unit of time
- Hence, counting the number of these operations performed by the algorithm will give as the total time required to execute the algorithm.

Example: Insertion Sort

```
procedure insertionSort(A : list of sortable items)
 for j = 2 to length [A]
   do key = A[j]
      // Insert A[j] into the sorted sequence A[1... j-1
      i = j - 1
      while i > 0 and A[i] > key
         do A[i + 1] = A[i]
         i = i - 1
      end while
      A[i + 1] = key
   end for
end procedure
```

Example: Notes

- Let's analyze the time complexity of insertion sort with an implementation presented above
- The time taken by the insertion sort procedure depends on
  - **Input size**: sorting a thousand numbers takes longer than ten
  - **Input sort status**: sorting an input already nearly sorted would be faster than a reversely sorted input
- We need to describe running time as a function of input size
- The notion of input size depends on the problem being studied
  - For many problems it is the number of items in the input
  - For others such as multiplying integers the best measure is the total number of bits
  - In other cases such as graph problems the input will described by more than one number
- We shall indicate which input size measure is being used with each problem we study

Example: Insertion Sort

- The running time of an algorithm on a particular input is the number of primitive operations or steps executed.
- It is convenient to define the notion of steps to make it machine-independent.
- It take a constant time to execute each line of our pseudocode but one line may take a different amount of time that the other.
- Let say line i of the pseudocode takes c_i to execute and the inner while loop tests t_j times for the j^{th} item.

Example: Insertion Sort

	INSERTIO-SORT (A)	Cost	Times
1	**for** j ← 2 **to** length [A]	C_1	N
2	**do** key ← A[j]	C_2	N-1
3	// Insert A[j] into the sorted sequence A[1... j-1]	0	N-1
4	i ← j – 1	C_4	N-1
5	**while** i > 0 **and** A[i] > key	C_5	$\sum_{j=2}^{n} t_j$
6	**do** A[I + 1] ← A[i]	C_6	$\sum_{j=2}^{n} t_j - 1$
7	j ← I – 1	C_7	$\sum_{j=2}^{n} t_j - 1 - 1$
8	A[i+1] ← key	C_8	N-1

Figure 2: Running Time Analysis of Insertion Sort

Example: Insertion Sort

- The total running time of the algorithm is the sum of running times of each statement executed

$$T(n) = C_1 n + C_2(n-1) + C_4(n-1) + C_5 sum_{j=2}^{n} t_j + C_6 sum_{j=2}^{n}(t_j - 1) + C_7 sum_{j=2}^{n}(t_j - 1) + C_8(n-1)$$

Example: Insertion Sort

- Running time of a program depends on the input size and in this case the level of sorting it already is in.
- For insertion sort the best case is when the input is already sorted in which case for each $j = 2, 3...n$ $A[i] < key$ in the first iteration of the while loop ($i = j - 1$) hence $t_j = 1$ leading to the following equation.

$$T(n) = C_1 n + C_2(n-1) + C_4(n-1) + C_5(n-1) + C_8(n-1)$$

$$T(n) = (C_1 + C_2 + C_4 + C_5 + C_8)n - (C_2 + C_4 + C_5 + C_8)$$

- Let's replace the constants

$$T(n) = An - B$$

Example: Insertion Sort

- If the array was in reverse order the algorithm will face its worst case results since we must compare each $A[j]$ with all the elements in the sorted portion of the array

$$\sum_{j=2}^{n} j = \frac{n(n+1)}{2} - 1$$

$$\sum_{j=2}^{n} j - 1 = \frac{n(n-1)}{2}$$

Example: Insertion Sort

$$T(n) = C_1 n + C_2(n-1) + C_4(n-1) + C_5(\frac{n(n-1)}{2} - 1) + C_6(\frac{n(n-1)}{2}) + C_7(\frac{n(n-1)}{2}) + C_8(n-1)$$

$$T(n) = (\frac{C_5}{2} + \frac{C_6}{2} + \frac{C_7}{2})n^2 + (C_1 + C_2 + C_4 + \frac{C_5}{2} - \frac{C_6}{2} + \frac{C_7}{2} + C_8)n - (C_2 + C_4 + C_5 + C_8)$$

- Let's replace the constants

$$T(n) = An^2 + Bn - C$$

Example: Insertion Sort

- So we can express the worst case running time as

$$T(n) = An^2 + Bn + C$$

- We used some simplifying abstractions to ease our analysis by ignoring actual statement cost and even the abstracts costs C_i.
- We normally even further simplify this by only considering **rate of growth** or **order of growth** instead of the actual running time.
- We do this because what we really want to know is how our algorithm performance compared to others for **large input sizes**.

Example: Insertion Sort

- We only consider the leading term An^2 since the lower order terms
  - Bn and C will have relatively insignificant for the large values of n compared to the first once.
- We even ignore the coefficient a due to similar reasons as to why we ignored the other terms to be left with n^2
- We consider n^2 as our order of growth.
- We call this the theta notation and it is presented as $\Theta(n^2)$
- We will return to the idea of asymptotic notation in a bit

Correctness of An Algorithm

Introduction

- In addition to analyzing the resource requirement one should proof its correctness.
- Proving the correctness means that showing that the algorithm **halts** with **correct answer** for **all valid inputs**
- There are several methods to prove the correctness of an algorithm
- In this course the primary technique we will use for proving correctness of an algorithm is **mathematical induction**
- Another less formal way of proving is **Intuitive Reasoning**
  - It explains why the algorithm works using natural language and logical reasoning.
  - This method is often the first step in proving correctness, providing a high-level understanding of the algorithm.

Introduction: Key Concepts

- Two key Concepts
  - **Inductive Proofs**: Use mathematical induction to prove correctness
  - **Loop Invariant**: Identify a property (invariant) that holds before and after each iteration of a loop.
  - Steps
    - **Initialization**: Show the invariant holds before the first iteration.
    - **Maintenance**: Show that if the invariant holds before an iteration, it holds after the iteration.
    - **Termination**: Show that when the loop terminates, the invariant and the loop condition together imply the desired result

Loop Invariant: Definition

- A loop invariant is a property of a program loop that is true before (and after) each iteration.
- It is a logical assertion, sometimes checked within the ode by an assertion call.
- Knowing its invariant(s) is essential in understanding the effect of a loop.
- It is what we will use to prove in the base case and the inductive step.
- We use the loop invariant to help us understand why an algorithm is **correct**.

Loop Invariant: Example

- In the pseudocode of insertion sort presented earlier, j indicates the current card being inserted.
- At the beginning of the outer loop (i.e., for loop), which is indexed by j, the sub array containing elements $A[1...j-1]$ consists of the currently sorted hand and elements $A[j+1...n]$ correspond to the pile of cards still on the table.
- Elements $A[1...j-1]$ are the elements originally positions 1 through $j-1$ but now in a sorted order.
- We state these properties formally as **loop invariants**

  At the start of each iteration of the for loop of lines 2-8, the subarray $A[1...j-1]$ *consists of the elements originally in* $A[1..j-1]$ **but in a sorted order**

Correctness Proof

- We must show three things about a loop invariant to proof an algorithm's correctness
  - **Initialization**: It is true prior to the first iteration of the loop
  - **Maintenance**: If it is true before an iteration of the loop, it remains true before the next iteration
  - **Termination**: when the loop terminates, the invariant gives us a useful property that helps show that the algorithm is correct
- Notice the similarity of this approach to mathematical induction

Correctness Proof: Similarity to Mathematical Induction

- In mathematical induction we need to proof the **base case** and then the **inductive step**.
- In proving correctness of algorithms
  - Showing the loop invariant holds for the first iteration is the base case
  - Showing that the invariant holds from iteration to iteration is like the inductive case.
- One important difference from mathematical induction is the third step in which we show the correctness for the termination case.
- In mathematical induction we show that the inductive step holds infinitely, here we stop the induction when the loop terminates.

Example: Insertion Sort

```
procedure insertionSort(A : list of sortable items)
 for j = 2 to length [A]
   do key = A[j]
      // Insert A[j] into the sorted sequence A[1... j-1]
      i = j - 1
      while i > 0 and A[i] > key
         do A[i + 1] = A[i]
         i = i - 1
      end while
      A[i + 1] = key
   end for
end procedure
```

Insertion Sort: Card Demo — Dance Demo

Example: Insertion Sort Proof

- **Loop Invariant**: At the j^{th} iteration the elements $[1...j-1]$ are sorted relative to themselves.

Example: Insertion Sort Proof

- **Loop Invariant**: At the j^{th} iteration the elements $[1...j-1]$ are sorted relative to themselves.
- **Initialization**:
  - We start by showing that the loop invariant holds before the first loop iteration when $j = 2$.
  - The subarray $[1...j-1]$, therefore, consists of just the single element $A[1]$, which in fact it the original element $A[1]$.
  - This array is sorted (trivial)

Example: Insertion Sort Proof

- **Maintenance**:

Example: Insertion Sort Proof

- **Maintenance**:
  - In the algorithm the for loop works by moving $A[j-1],\ A[j-2],\ A[j-3]$ and so on by one position to the right until the right position for $A[j]$ is found at which point the value of $A[j]$ is inserted.
  - More formally, though, it is required to show the loop invariant of the inner loop as well.
  - Here, however, we will not go into such detail for the time being

Example: Insertion Sort Proof

- **Maintenance**:
  - In the algorithm the for loop works by moving $A[j-1],\ A[j-2],\ A[j-3]$ and so on by one position to the right until the right position for $A[j]$ is found at which point the value of $A[j]$ is inserted.
  - More formally, though, it is required to show the loop invariant of the inner loop as well.
  - Here, however, we will not go into such detail for the time being
- **Termination**:

Example: Insertion Sort Proof

- **Maintenance**:
  - In the algorithm the for loop works by moving $A[j-1],\ A[j-2],\ A[j-3]$ and so on by one position to the right until the right position for $A[j]$ is found at which point the value of $A[j]$ is inserted.
  - More formally, though, it is required to show the loop invariant of the inner loop as well.
  - Here, however, we will not go into such detail for the time being
- **Termination**:
  - The outer loop end when j exceeds n
  - This is when $j = n + 1$ substituting $n + 1$ for j in the wording of loop invariant we have the $subarray A[1...n]$ consists of the elements originally in $A[1..n]$ but in a sorted order.
  - But the subarray $A[1...n]$ is the entire array which means the entire array is sorted

Example: Bubble Sort

```
procedure bubbleSort(A)
  for i = n−1 down to 0 do
    for j = 0 to i do
      if A[j+1] < A[j] then
        tmp = A[j+1]
        A[j+1] = A[j]
        A[j] = tmp
      end if
    end for
  end for
end procedure
```

Insertion Sort: Card Demo — Dance Demo

Example: Bubble Sort Proof

- **Loop Invariant**: At the i^{th} iteration the last i elements $[n-i, ...n]$ are sorted relative to themselves and are all greater than all elements in $[1, ...n-i-1]$.
- **Initialization**:
  - Before the first loop iteration $i = n-1$.
  - The subarray $[n-i...n]$ consists of no element as $n-n-1 = n$, and $A[n]$ is empty as we are using 0 based index.
  - This array is sorted (trivial)

Example: Bubble Sort Proof

- **Maintenance**:
  - In the algorithm the inner for loop bubbles the maximum element from $A[0],\ A[1],\ A[2],\ \ldots\ A[i]$ to the end (i.e., the i^{th} position.
  - This means if we assume the elements i *to* n are sorted and they are all greater than all elements $0\ -i\ -\ 1$ in the i^{th} iteration, then on the so on $(i\ +\ 1)^{th}$ iteration the biggest element from $A[0],\ A[1],\ A[2],\ \ldots\ A[i-1]$ will be on $(i\ -\ 1)^{th}$ location making the lats $(i\ +\ 1)$ sorted hence maintaining the loop invariant.
- **Termination**:
  - The outer loop end when i becomes 0 hence from the loop invariant we have when the loop terminates the last $n-0$ elements are sorted which means the entire list is sorted.

Rate of Growth

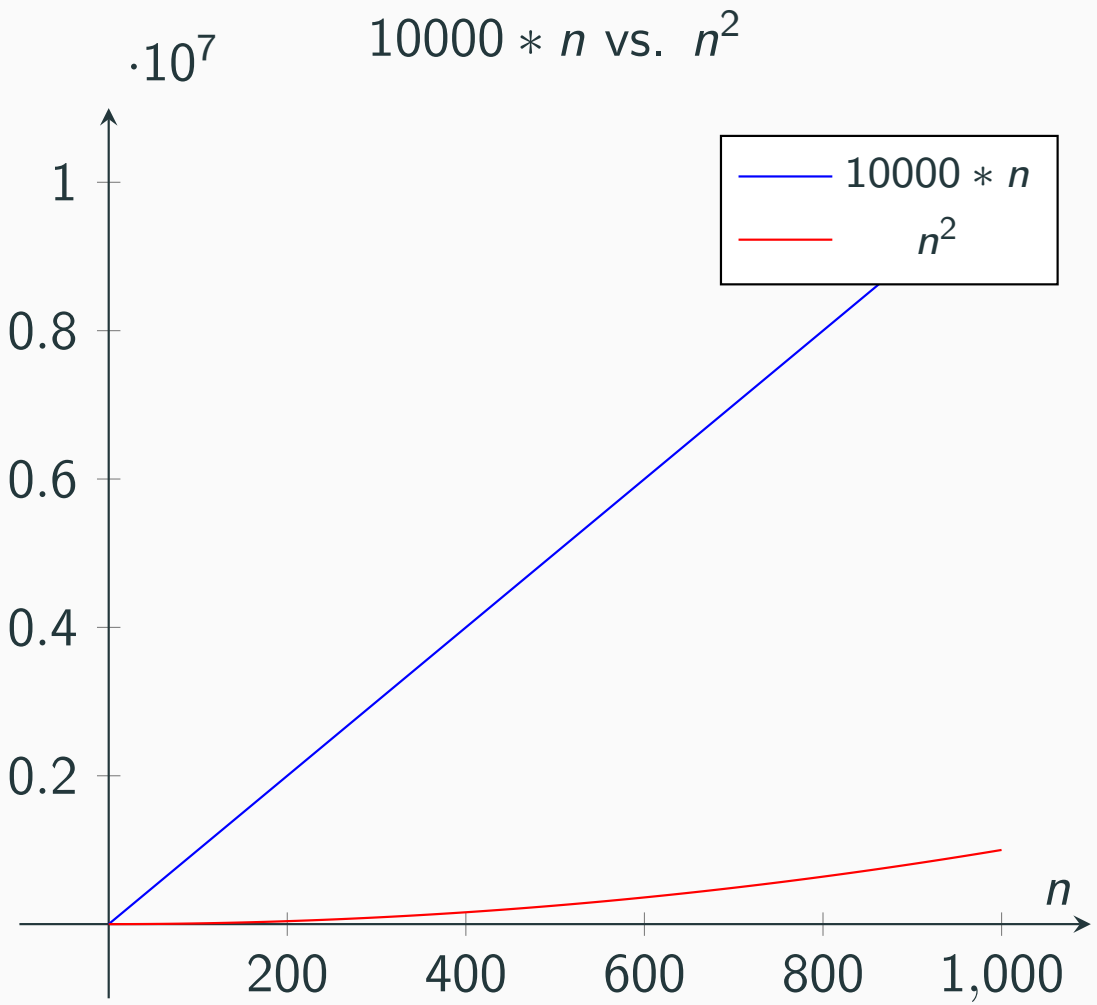
Comparing Mathematical Functions

- Assume all (mathematical) functions take only positive arguments and values.
- Different algorithms for the same problem may have different (worst-case) running times.
- Example of sorting: bubble sort, insertion sort, quick sort, merge sort, etc.

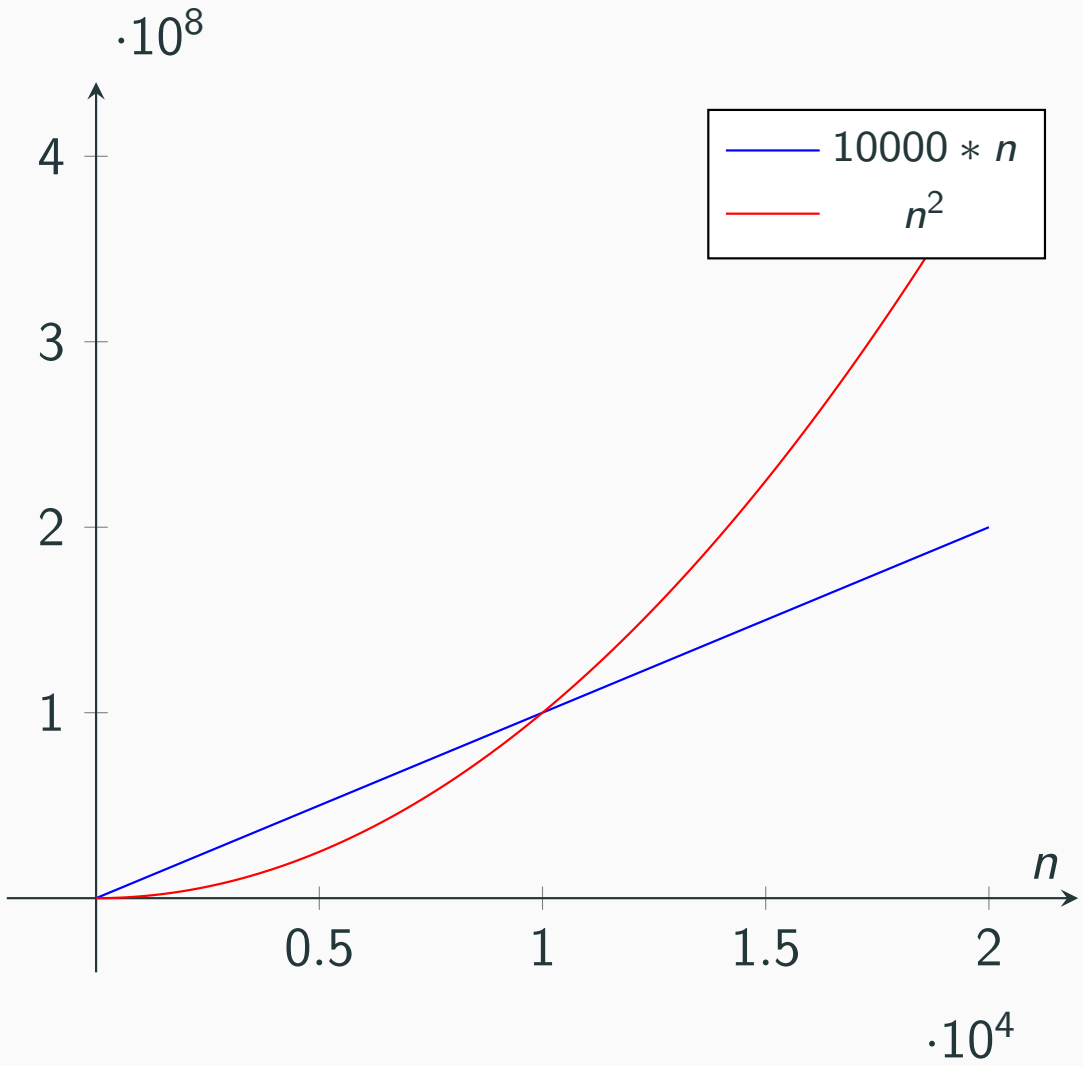
Comparing Mathematical Functions

- Assume all (mathematical) functions take only positive arguments and values.
- Different algorithms for the same problem may have different (worst-case) running times.
- Example of sorting: bubble sort, insertion sort, quick sort, merge sort, etc.
- Bubble sort and insertion sort take roughly n^2 comparisons while quick sort (only on average) and merge sort take roughly $n \log_2 n$ comparisons.
  - "Roughly" hides potentially large constants, e.g., running time of merge sort may in reality be $10 n log_2 n$.
- How can make statements such as the following, in order to compare the running times of different algorithms?
  - $100 n log_2 n \leq n^2$
  - $10000 n \leq n^2$
  - $5n^2 4n \geq 1000 n log n$

Comparing Mathematical Functions



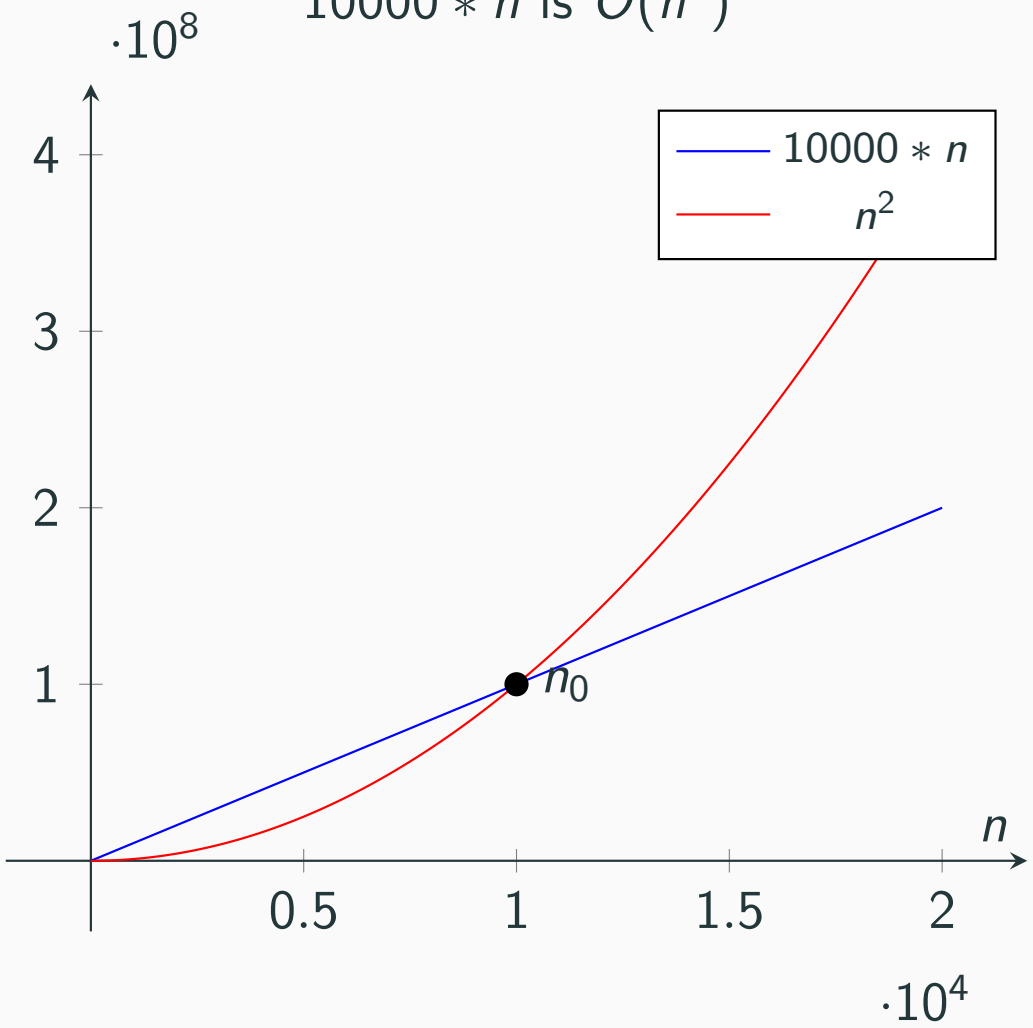
Comparing Mathematical Functions



Upper Bound: Definition

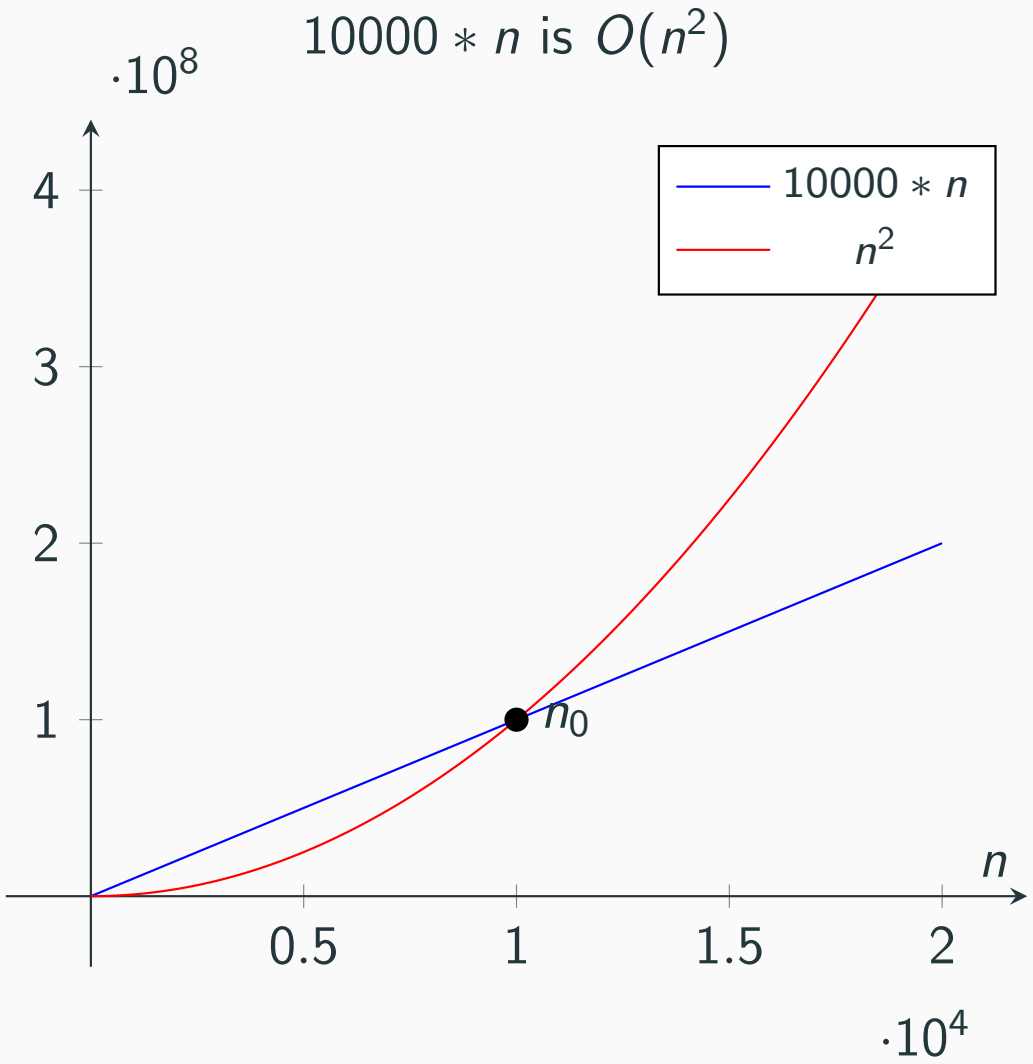
Asymptotic upper bound: A function $f(n)$ is $O(g(n))$ if there exists a constant $c > 0$ and $n_0 \geq 0$ such that for all $n \geq n_0$ $f(n) \leq cg(n)$

$10000 * n$ is $O(n^2)$



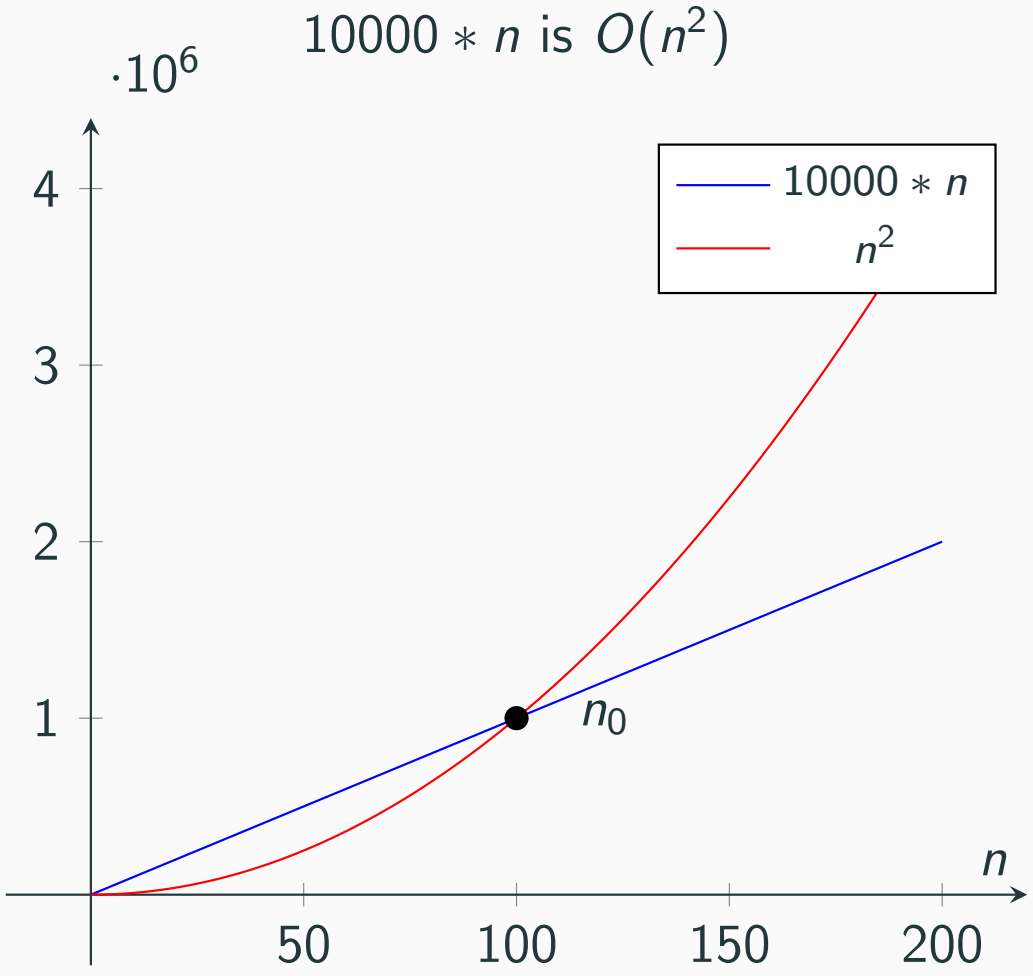
Upper Bound: Example

- If $c = 1$ then $n_o = 10^3$



Upper Bound: Example

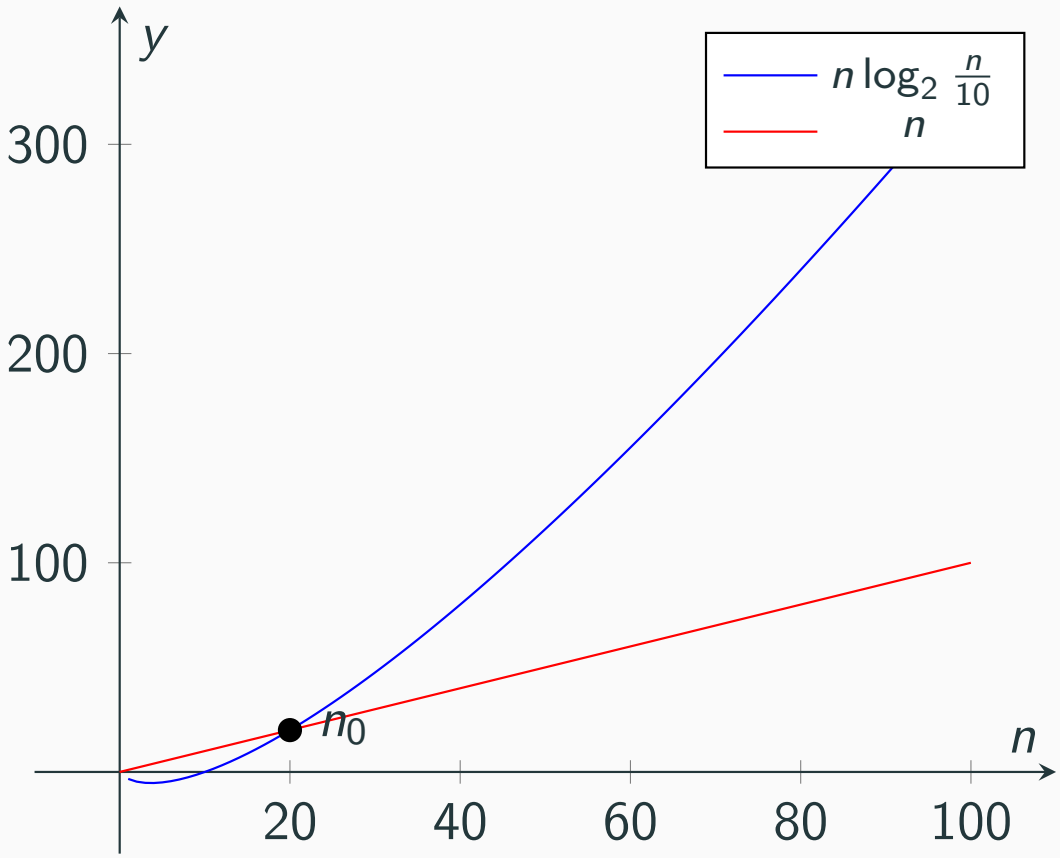
- If $c = 100$ then $n_o = 100$



Lower Bound: Definition

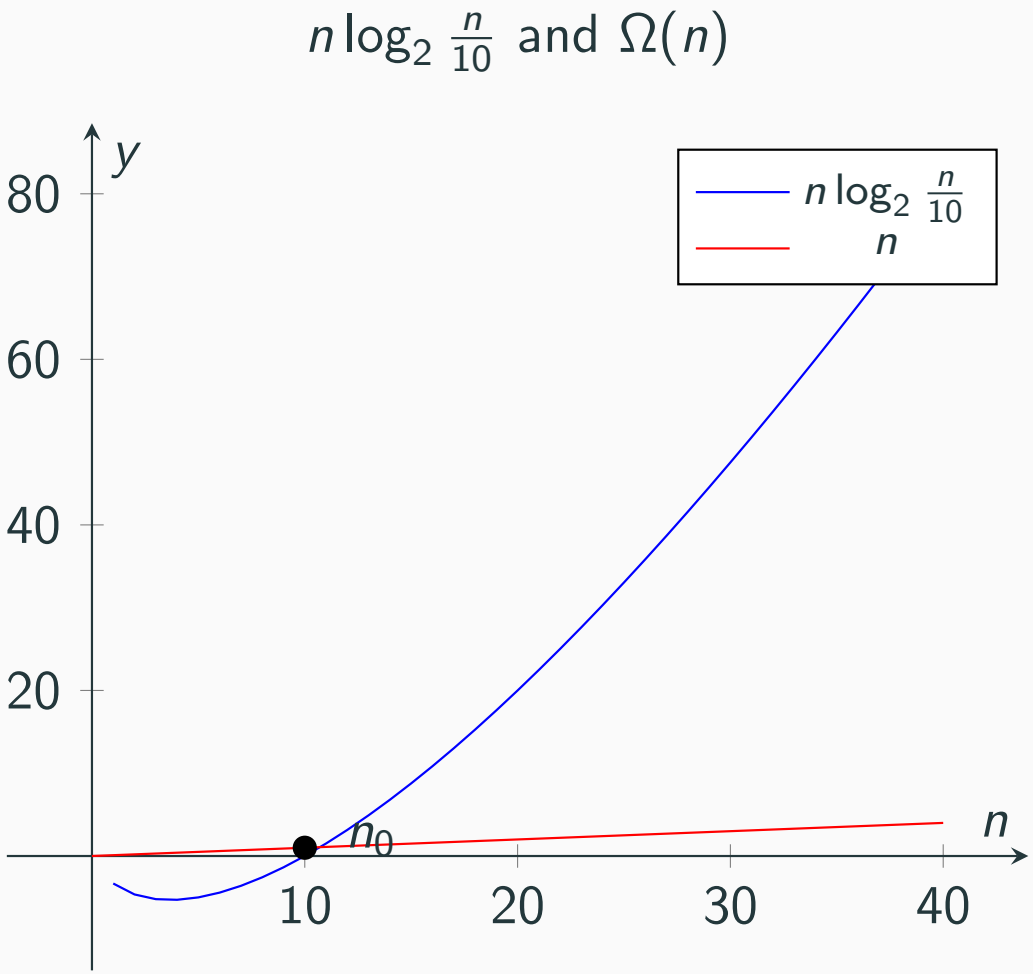
Asymptotic lower bound: A function $f(n)$ is $\Omega(g(n))$ if there exists a constant $c > 0$ and $n_0 \geq 0$ such that for all $n \geq n_0$ $f(n) \geq cg(n)$

$n \log_2 \frac{n}{10}$ and $\Omega(n)$



Lower Bound: Example

- If $c = 1/10$ then $n_o = 10$



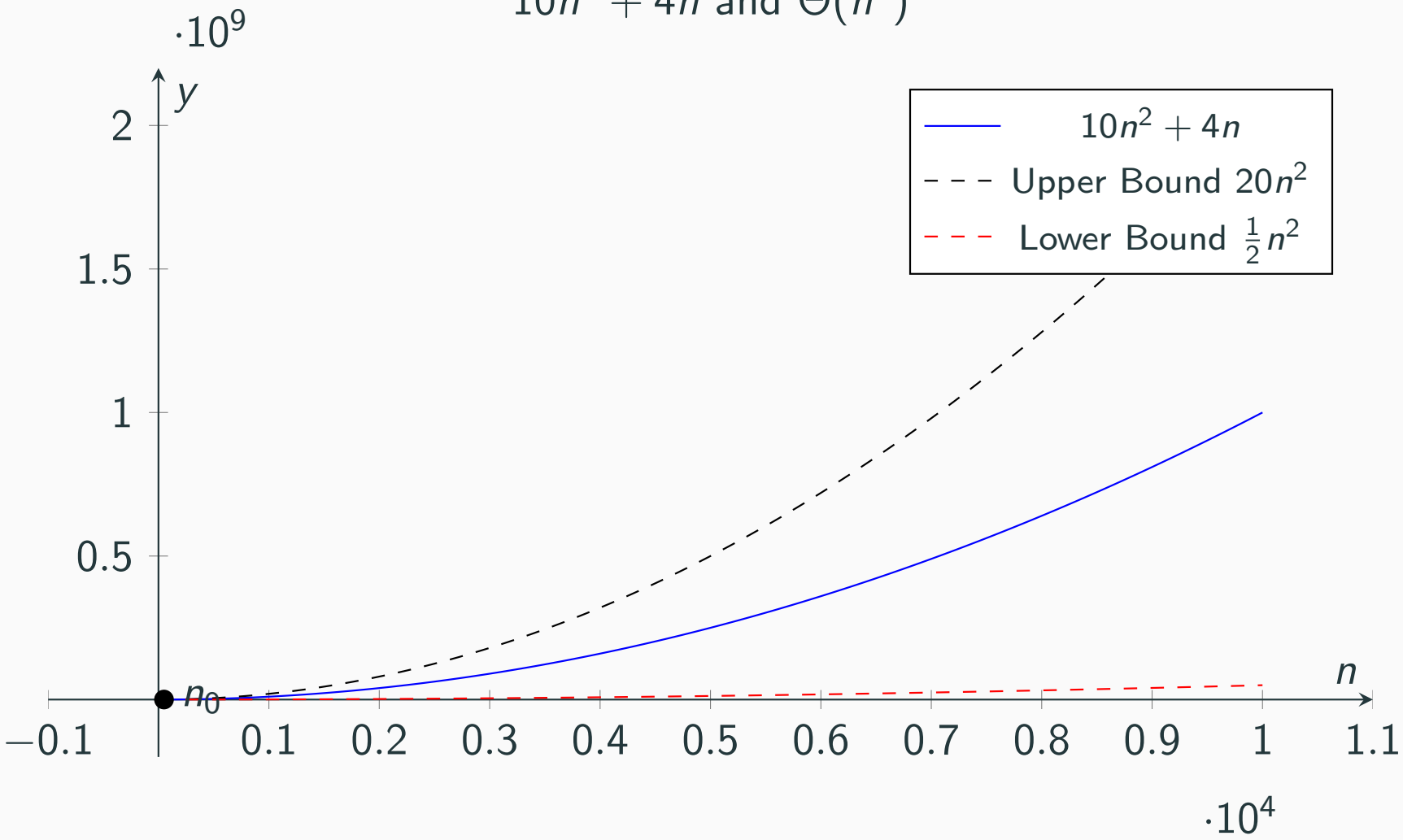
Lower Bound: Meanings in Different Contexts

- Mathematical functions: n is a lower bound for $nlogn/10$, i.e., $nlogn/10 = \Omega(n)$.
  - This statement is purely about these two functions.
  - Not in the context of any algorithm or problem.
- Algorithms:
  - The lower bound on the running time of bubble sort is $\Omega(n^2)$
  - There is some input of n numbers that will cause bubble sort to take at least $\Omega(n^2)$ time
  - But there may be other, faster algorithms for sorting.
- Problems:
  - The problem of sorting n numbers has a lower bound of $\Omega(nlogn)$
  - For any comparison-based sorting algorithm, there is at least one input for which that algorithm will take $\Omega(nlogn)$ steps.
  - The stable matching problem has a lower bound of $\Omega(n^2)$

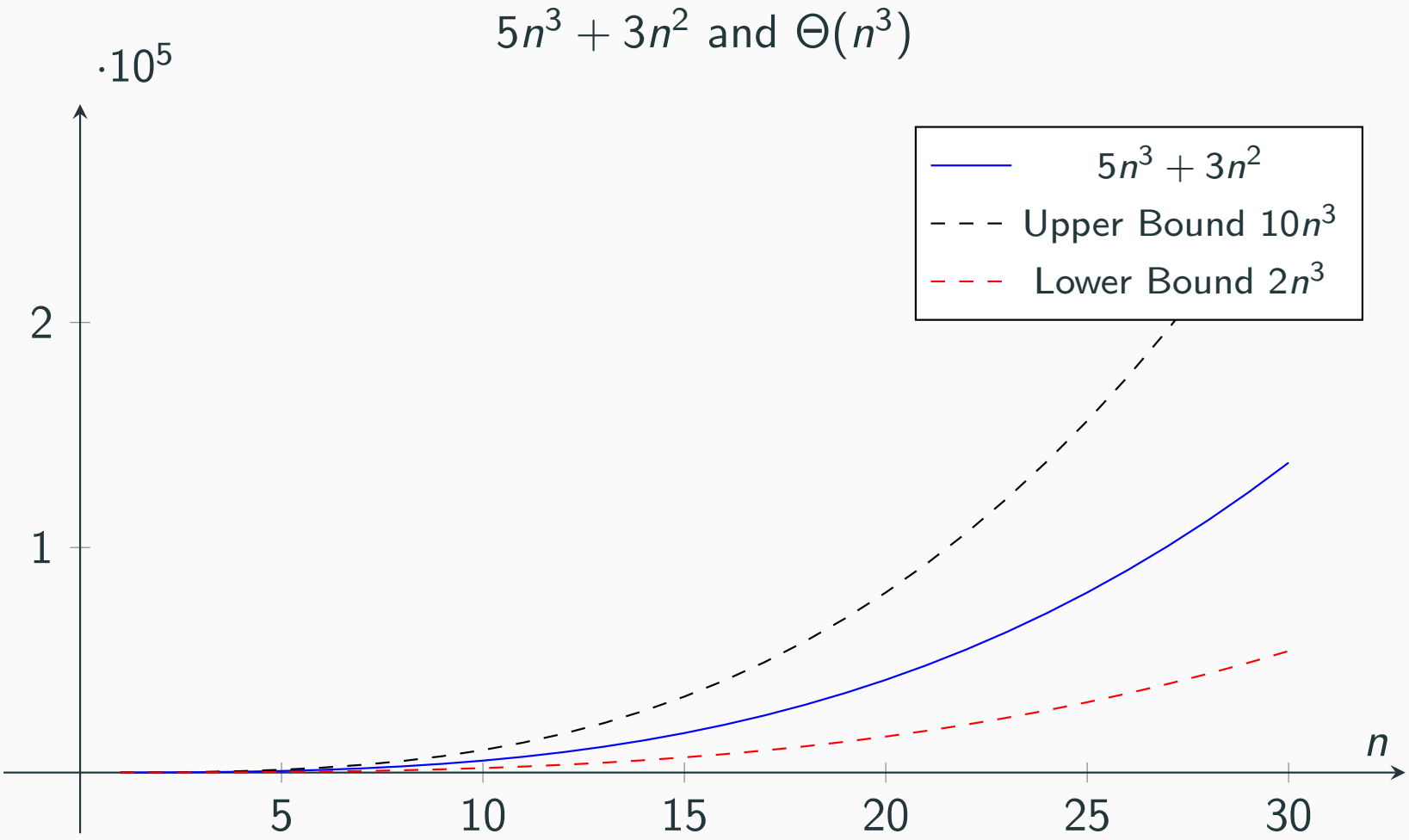
Tight Bound: Definition

Asymptotic lower bound: A function $f(n)$ is $\Theta(g(n))$ if $f(n)$ is $O(g(n))$ and $f(n)$ is $\Omega(g(n))$

$10n^2 + 4n$ and $\Theta(n^2)$



Tight Bound: Example



Properties of Asymptotic Growth Rates

- Transitivity
  - If $f = O(g)$ and $g = O(h)$, then $f = O(h)$.
  - If $f = \Omega(g)$ and $g = \Omega(h)$, then $f = \Omega(h)$.
  - If $f = \Theta(g)$ and $g = \Theta(h)$, then $f = \Theta(h)$.

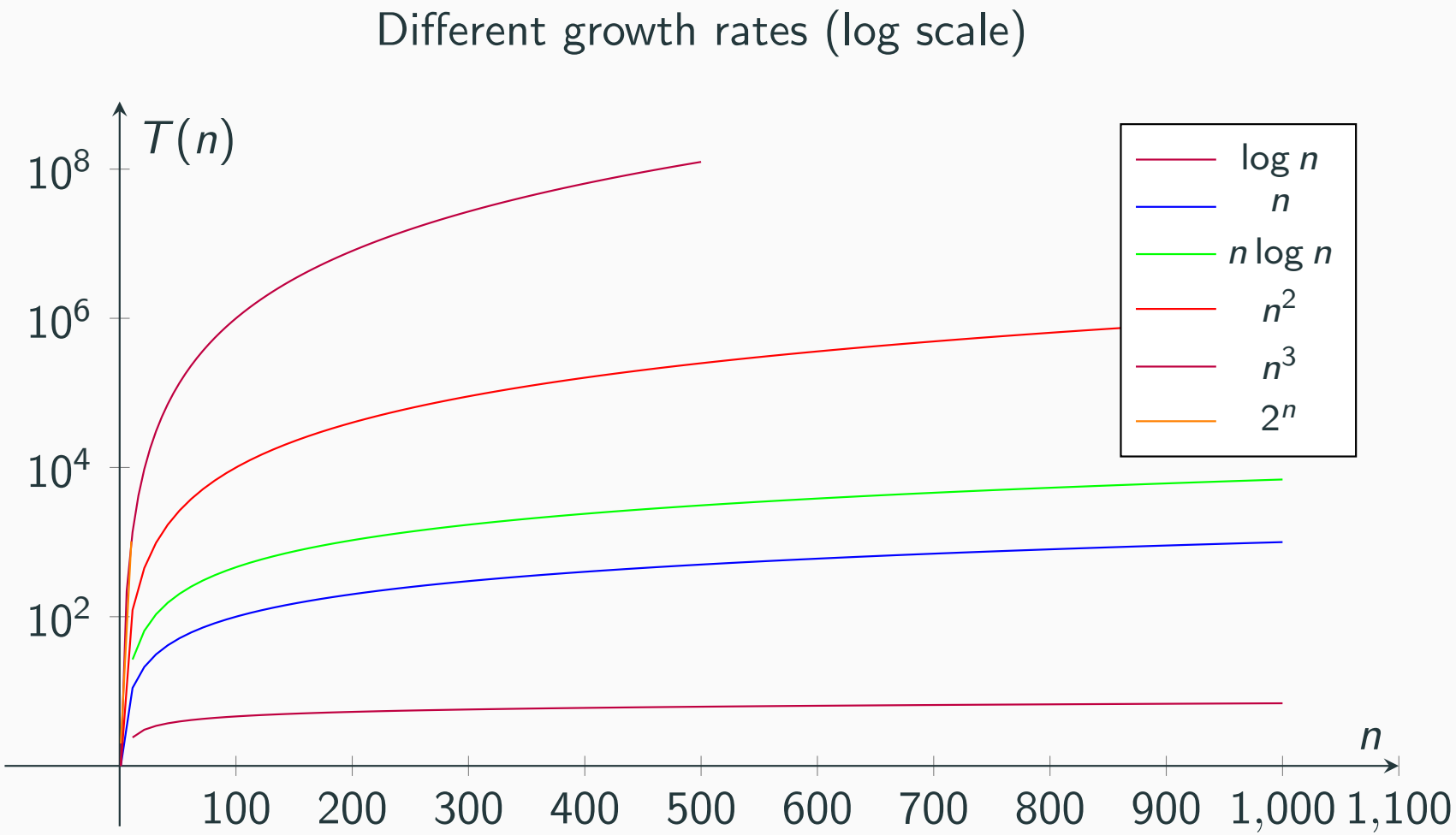
Properties of Asymptotic Growth Rates

- Transitivity
  - If $f = O(g)$ and $g = O(h)$, then $f = O(h)$.
  - If $f = \Omega(g)$ and $g = \Omega(h)$, then $f = \Omega(h)$.
  - If $f = \Theta(g)$ and $g = \Theta(h)$, then $f = \Theta(h)$.
- Additivity
  - If $f = O(h)$ and $g = O(h)$, then $f + g = O(h)$
  - Similar statements hold for lower and tight bounds
  - If $f = O(g)$, then $f + g = \Theta(g)$

Example Asymptotic Growth Rates

$f(n)$	$g(n)$	**Reason**
$pn^2 + qn + r$	$\Theta(n^2)$	
$pn^2 + qn + r$	$O(n^3)$	$n^2 \leq n^3$, if $n \geq 1$
$\sum_{0 \leq i \leq d} a_i n^i$	$\Theta(n^d)$	if $d > 0$ is an integer constant and $a_d > 0$
$\log_a n$	$O(\log_b n)$	For any pair of constants $a, b > 1$

Example Asymptotic Growth Rates



Example Problems: Searching and Sorting

- Searching
  - Linear Search : $\Theta(n)$
  - Binary Search : $\Theta(logn)$
- Sorting
  - Most efficient solution : $\Theta(nlogn)$
  - Insertion, Bubble, Selection: $\Theta(n^2)$
  - Merge Sort
    - Worst-case: $\Theta(nlogn)$
    - Average-case: $\Theta(nlogn)$
    - Best-case: $\Theta(nlogn)$
  - Quick Sort
    - Worst-case: $\Theta(n^2)$
    - Average-case: $\Theta(nlogn)$
    - Best-case: $\Theta(nlogn)$
- Other Problems
  - Stable Matching: $\Theta(n^2)$
  - 0-1 Knapsack: $\Theta(n^2)$
  - Fractional Knapsack: $\Theta(nlogn)$
    - The costliest step is sorting

Example Problems: Other Problems

- Stable Matching: $\Theta(n^2)$
- Knapsack:
  - 0-1 Knapsack: $\Theta(n^2)$
  - Fractional Knapsack: $\Theta(nlogn)$
    - The costliest step is sorting
- Graph Coloring
  - A method of assigning colors to the vertices of a graph such that no two adjacent vertices share the same color.
  - This problem has several applications in fields such as scheduling, register allocation in compilers, and frequency assignment in wireless networks.
  - Greedy Vertex Algorithm: $O(V + E)$
  - Backtracking Algorithm: Exponential $O(k^V)$
  - DSatur Algorithm: $O(V^2 logV + VE)$

Example Problems: Other Problems

- Matrix Multiplication
  - Time Complexity $O(n^3)$
  - We have a way to slightly improve this
- Solving the Traveling Salesman Problem (TSP) using brute-force
  - Time Complexity $O(2^n)$
- Generating all permutations of a string.
  - Given a string of length n, generating all possible permutations involves $n!$ operations
- Accessing an element in an array
  - Time Complexity $\Theta(1)$

Summary

Takeaway

- Algorithm Analysis
  - Means finding out the resource it take execute an algorithm
  - Proving it terminates with correct result
- Key concepts
  - Prove by induction
  - Loop invariant: Statements to prove over the three steps
- Comparing time complexity
  - Three Cases
    - Best Case
    - Average Case
    - Worst Case
  - Rate of growth
    - Big O
    - Theta
    - Big Omega

Things to know

- Algorithm Analysis
  - Identifying the loop invariant
  - Using the modified mathematical induction to prove correctness of an algorithm
    - Initialization
    - Maintenance
    - Termination
  - Identifying the best and worst case of an algorithm
    - Both when that occurs and the runtime
  - Rate of growth