CSI 2103: Data Structures

Algorithm Analysis (Ch 4)

Yonsei University

Spring 2022

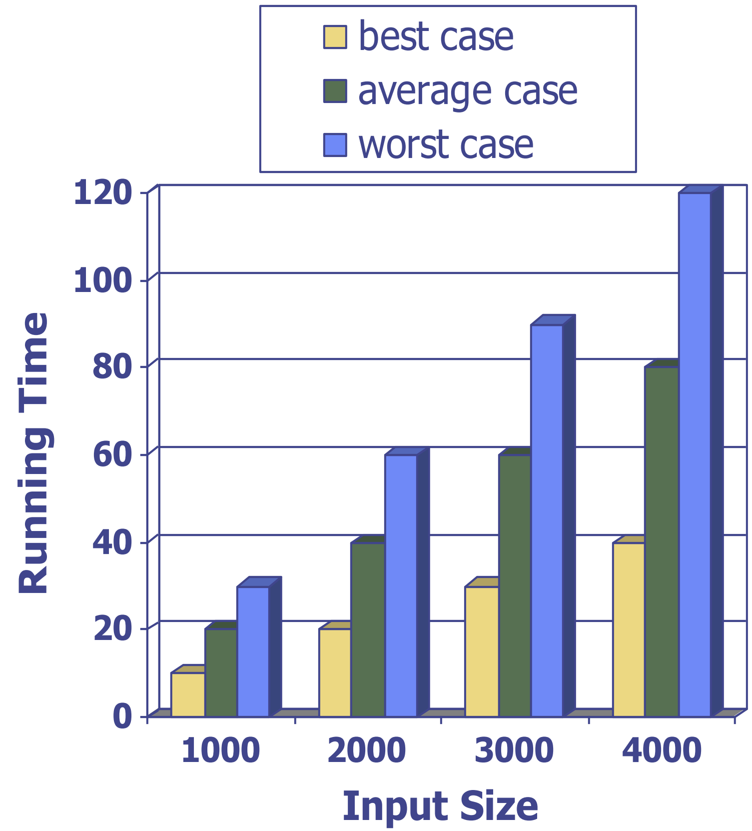
Seong Jae Hwang

Aims

- How do we define a "good" data structure?
- A "good" data structure is fast (efficient) at performing various operations as data grows:
  - Finding stuff
  - Adding stuff
  - Deleting stuff
- Operations are algorithms!
- How do we measure the speed (running time) of the operations/algorithms?

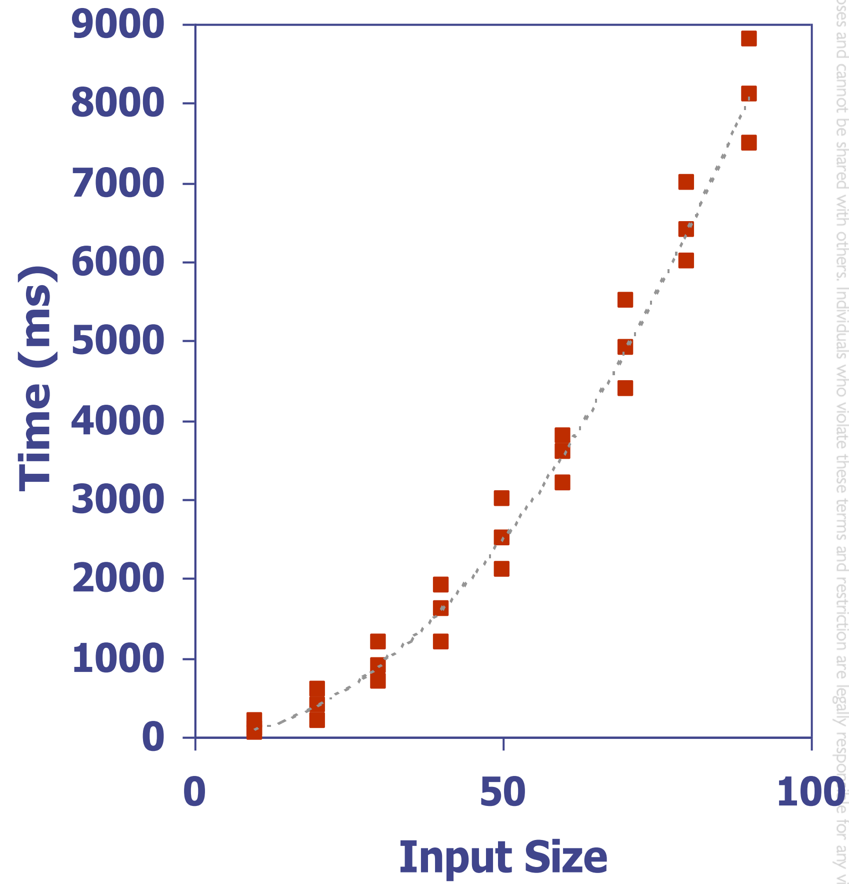
Running Time of Algorithms

- Algorithms
  - take inputs of varying input sizes n
  - take running time t to perform
- Running time t (usually) grows with the input size n
- Q: How do we measure t?
- Q: How fast does t grow as n grows?



Experimental Studies

- Implement the algorithm as a program
- Run the program with varying n and record the running time t
- Plot and observe



Measure the elapsed time of an algorithm

```
from time import time
start_time = time( )                    # record the starting time
run algorithm
end_time = time( )                      # record the ending time
elapsed = end_time − start_time         # compute the elapsed time
```

Example 1

- Two algorithms (repeat1 and repeat2) of concatenating a character multiple times:
    - in Java (no need to understand how they work)

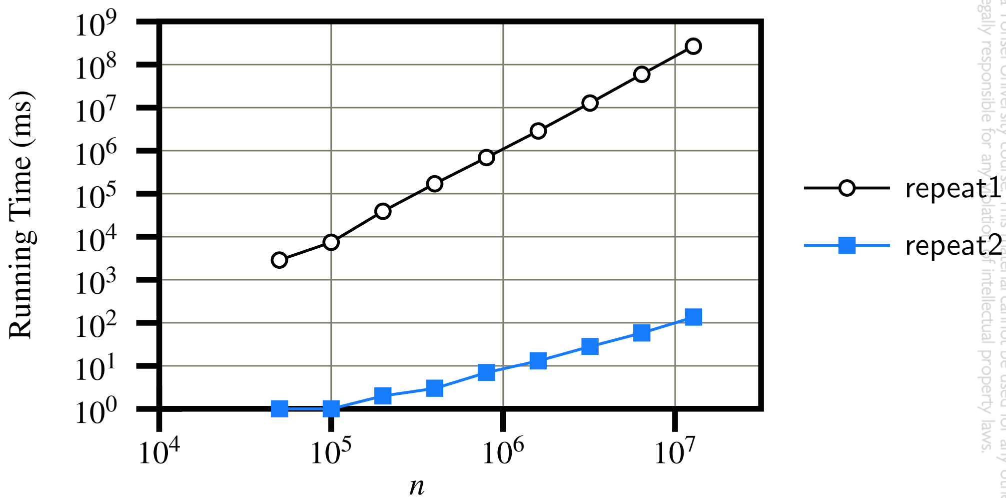
```
/** Uses repeated concatenation to compose a String with n copies of character c. */
public static String repeat1(char c, int n) {
  String answer = "";
  for (int j=0; j < n; j++)
    answer += c;
  return answer;
}

/** Uses StringBuilder to compose a String with n copies of character c. */
public static String repeat2(char c, int n) {
  StringBuilder sb = new StringBuilder( );
  for (int j=0; j < n; j++)
    sb.append(c);
  return sb.toString( );
}
```

Example 1

- Compare the elapsed times (milliseconds):
- For $n = 12.8M$
  - repeat1 takes 3 days
  - repeat2 takes 135 ms
- When n doubles
  - repeat1's t increases x4
  - repeat2's t increases x2
- Just like these functions, operations in data structures also depend on the "size"

n	repeat1 (in ms)	repeat2 (in ms)
50,000	2,884	1
100,000	7,437	1
200,000	39,158	2
400,000	170,173	3
800,000	690,836	7
1,600,000	2,874,968	13
3,200,000	12,809,631	28
6,400,000	59,594,275	58
12,800,000	265,696,421	135



Limitations of Experimental Studies

- 3 major limitations:
  - Experimental running times of two algorithms need the exact same hardware and software environments
  - Each n needs to be explicitly tested to measure corresponding running times.
    - E.g.: Do we want to try $n = 100B$?
    - E.g.: Technically, $n = 75K$ was not tested, so we will never know!
  - An algorithm must be fully implemented to test it!
    - Very impractical

n	repeat1 (in ms)	repeat2 (in ms)
50,000	2,884	1
100,000	7,437	1
200,000	39,158	2
400,000	170,173	3
800,000	690,836	7
1,600,000	2,874,968	13
3,200,000	12,809,631	28
6,400,000	59,594,275	58
12,800,000	265,696,421	135

Theoretical Analysis

- How can we analyze the algorithm efficiency such that
  - We can compare algorithms independent of hardware and software environment?
  - We can consider all possible n?
  - We do not need to implement and actually run the algorithm with a program?
- In other words: can we approximate how the running time t grows as the input size n grows just by "looking at the code"?

Step 1: Define Primitive Operations

- Very basic computations
- Largely independent from the programming language
- Practically, these are all very fast and take similar amount of time to perform
  - Assigning a value to a variable
  - Following an object reference
  - One arithmetic operations (e.g., 1+1)
  - Comparing two numbers
  - Accessing an element of an array by index
  - Calling a method
  - Returning from a method
- t simply "counts" these operations as a new "unit" of running time

Example: Count the Ops

- Finding max element of an array of size n
- Operations per line:
  - L3: 2 ops
  - L4: $2n$ ops
  - L5: $2n$ ops
  - L6: 0 to n ops // what is going on here?
  - L7: 1 op

```
1 def find_max(data):
2     """Return the maximum element from a nonempty Python list."""
3     biggest = data[0]                 # The initial value to beat
4     for val in data:                  # For each value:
5         if val > biggest              # if it is greater than the best so far,
6             biggest = val             # we have found a new best (so far)
7     return biggest                    # When loop ends, biggest is the max
```

Example: Running Time May Vary

- Worst case: $5n + 3$ ops
- Best case: $4n + 3$ ops
- Define
  - a = time taken by the fastest op
  - b = time taken by the slowest op
- Let $T(n)$ be the worst-case time of find_max. Then,

$$a(4n + 3) \leq T(n) \leq b(5n + 3)$$

- Running time $T(n)$ is bounded by two linear functions

Example: Too Precise

$$a(4n + 3) \leq T(n) \leq b(5n + 3)$$

- Different environments (hardware, software, implementation, etc.)
  - affects $T(n)$ by a constant factor (e.g., different a and b)
  - but does not change the growth rate of $T(n)$ with respect to n
- Why worst-case analysis?
  - The running times may be different with the same everything!
    - $4n + 3$ vs. $5n + 3$
  - Average depends on the distribution of the inputs
- In general, the running time analysis involves the worst-case with respect to n
  - "prepare for the worst": every possible n will do at least as well as the analysis

Example: Focus on what's important

$$a(4n+3) \leq T(n) \leq b(5n+3)$$

- What really matters is the relationship between the running time t and the input size n
  - Constants usually do not matter much
    - As n grows, the constant +5 becomes meaningless
  - Exact ops time (a and b) do not matter much
  - How the worst-case running time is bounded from above
- The worst-case running time $T(n)$ has the linear growth rate with respect to n
- Will be more formal later. Are there other common growth rates?

Growth Rates

- Let $f(n)$ be the running time as a function of n
- 7 common growth rates (slowest to fastest):

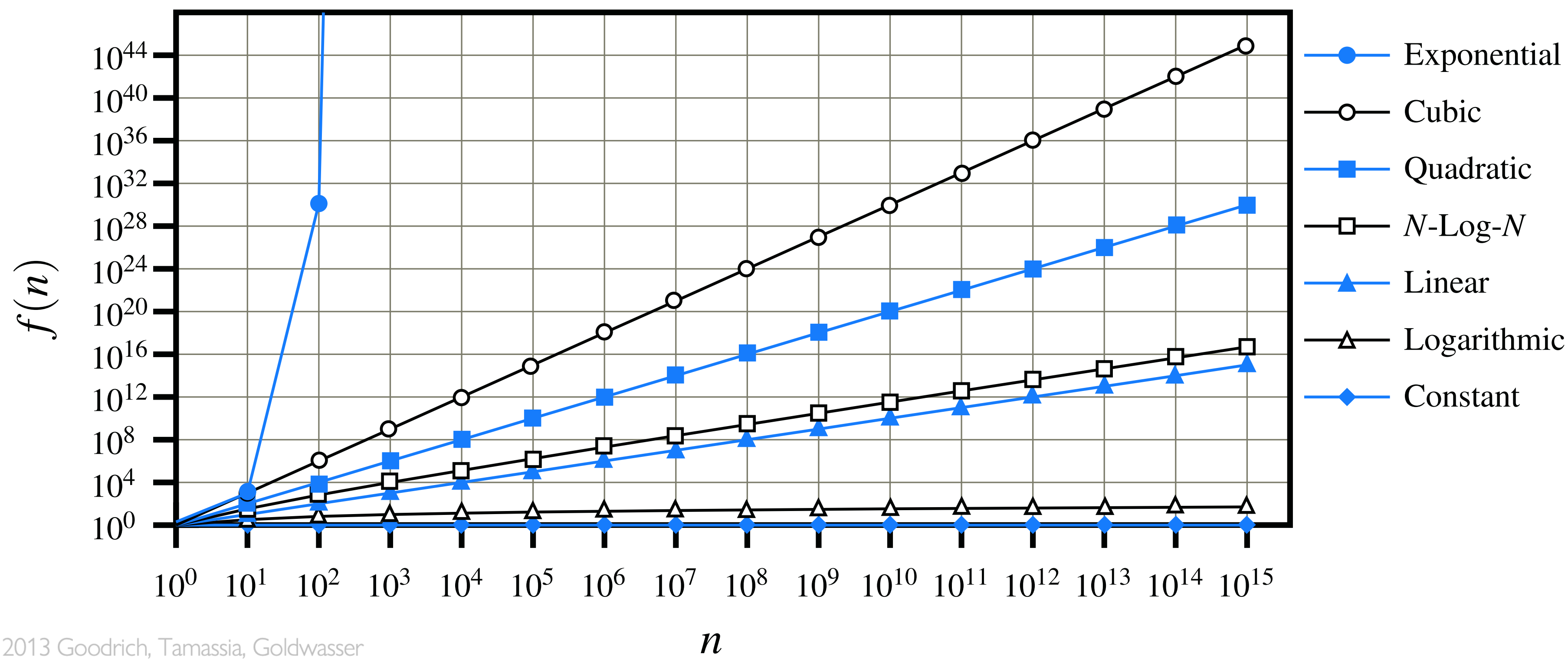
constant	**logarithm**	**linear**	***n*-log-*n***	**quadratic**	**cubic**	**exponential**
1	$\log n$	n	$n \log n$	n^2	n^3	a^n

- In this class, usually $\log_2 n = \log n$
- Recall repeat1 and repeat2: their $f(n)$?

n	repeat1 (in ms)	repeat2 (in ms)
50,000	2,884	1
100,000	7,437	1
200,000	39,158	2
400,000	170,173	3
800,000	690,836	7
1,600,000	2,874,968	13
3,200,000	12,809,631	28
6,400,000	59,594,275	58
12,800,000	265,696,421	135

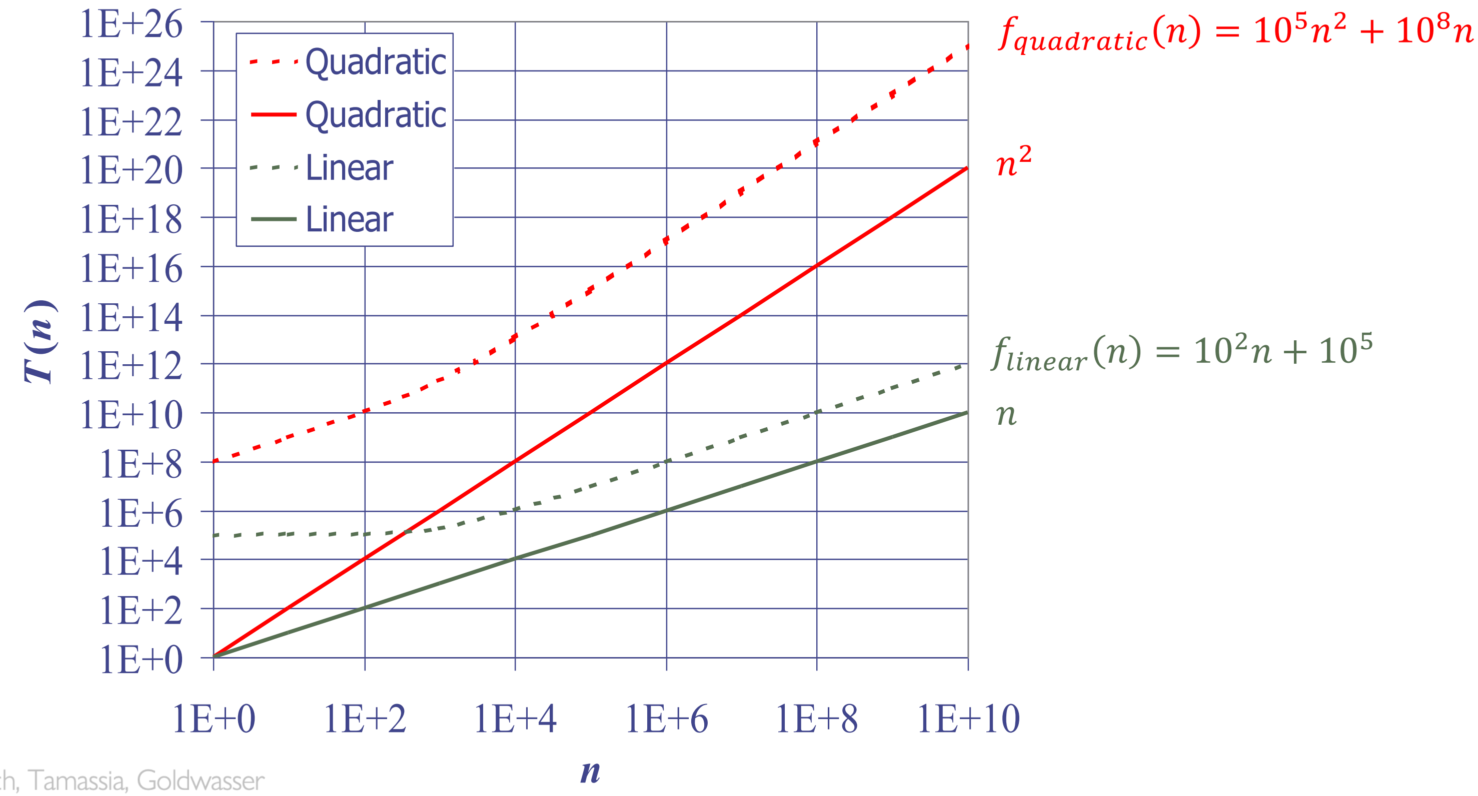
Growth Rate Matters

- As n grows larger and larger, the growth rates matters more and more!
- How do we see the "big picture" as n grows?
  - Does $4n$ vs. $5n$ matter compared to $40000n$ vs. n^2?



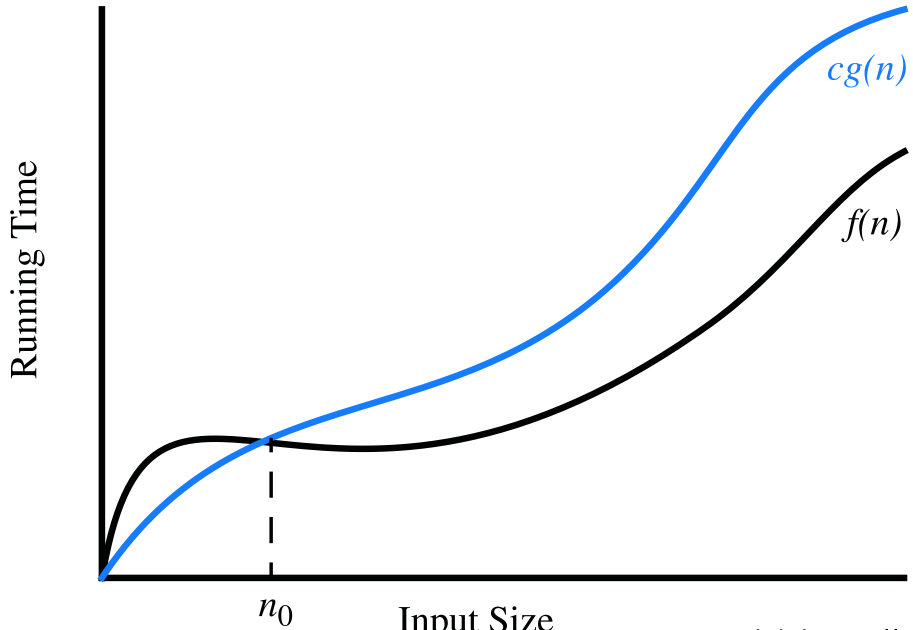
Asymptotic Analysis: $n \rightarrow \infty$

- We will "simplify" the running times to meaningfully compare them as n grows infinitely
  - E.g., Essentially, $4n \approx 5n$
  - E.g., As n grows, $40000n \ll n^2$
- Which term "dominates" the speed of growth?



Order of Growth: Big-Oh

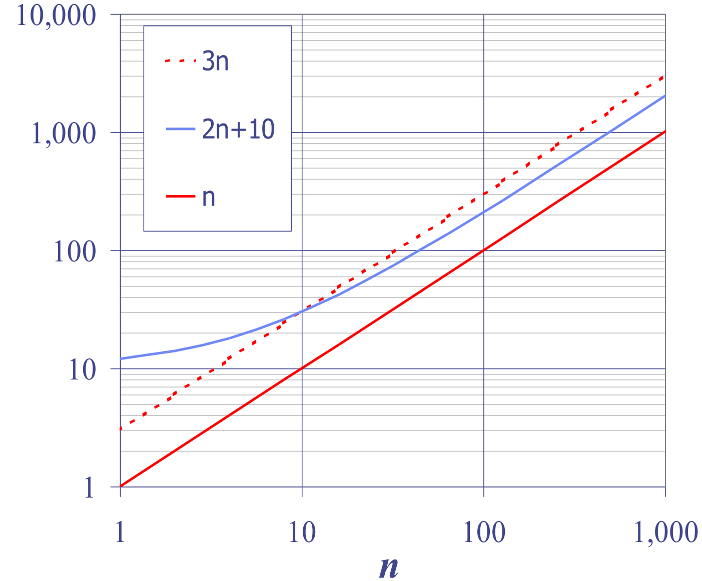
- We want to formally say "the growth rate $f(n) = 4n$ is essentially the same as a simple linear growth rate $g(n) = n$"
- If $f(n)$ can be bounded by $g(n)$ from above after multiplying $g(n)$ by a constant, we say "$f(n)$ is big-Oh of $g(n)$"
- Formally: Given $f, g: \mathbb{N} \to \mathbb{R}^+$, if there are constants $c > 0$ and $n_0 \geq 1$ such that $f(n) \leq c \cdot g(n)$ for all $n \geq n_0$, then we say that $f(n)$ is $O(g(n))$.



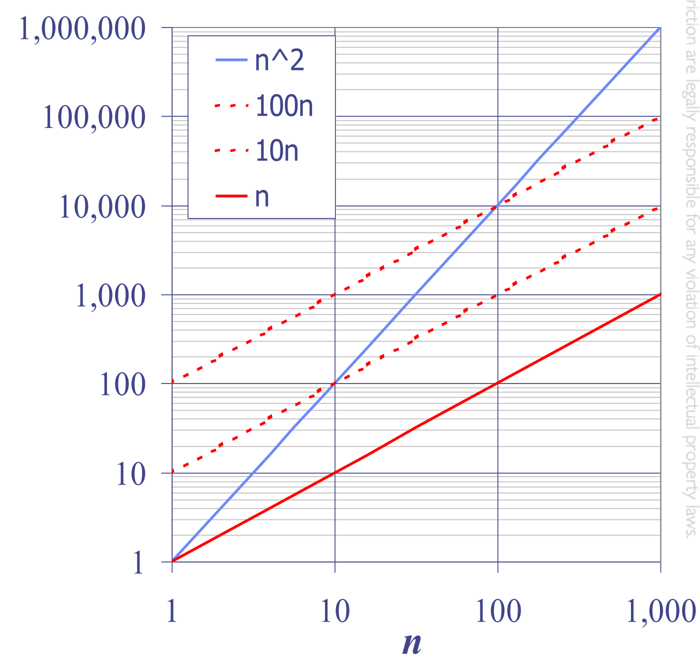
We will also allow $f(n) = O(g(n))$

Examples

- $2n + 10$ is $O(n)$ for $c = 3, n_0 = 10$
- $n + 1$ is $O(n)$ for $c = 2, n_0 = 1$
- $8n + 5$ is $O(n)$ for $c = 9, n_0 = 5$



- Intuitively, as n grows, polynomial functions will be dominated by its degree.
- $5n^4 + 3n^3 + 2n^2 + 4n + 1$ is $O(n^4)$ for $c = 15, n_0 = 1$



- n^2 is not $O(n)$ since $n^2 \leq cn$ cannot be satisfied by a constant c

Examples

- $3\log n + 2$ is $O(\log n)$ for $c = 4, n_0 = 2$
- $2^{n+2} = O(2^n)$ for $c = 4, n_0 = 1$
- $2n + 100\log n = O(n)$ for $c = 102, n_0 = 1$

- Use the smallest possible class of functions:
  - $2n = O(n)$ instead of $2n = O(n^2)$
- Use the simplest expression of the class

Big-Oh Rules

- If $f(n)$ is a polynomial of degree d, then $f(n) = O(n^d)$
  - Drop lower-order terms
  - Drop constant factors
  - Ex: $5n^4 + 3n^3 + 2n^2 + 4n + 1 = O(n^4)$
- Use the smallest possible class of functions:
  - $2n = O(n)$ instead of $2n = O(n^2)$
- Use the simplest expression of the class
  - $3n + 5 = O(n)$ instead of $3n + 5 = O(3n)$

Big-Omega: "Opposite" of Big-Oh

- Big-Oh: Given $f, g: \mathbb{N} \to \mathbb{R}^+$, if there are constants $c > 0$ and $n_0 \geq 1$ such that $f(n) \leq c \cdot g(n)$ for all $n \geq n_0$, then we say that $f(n)$ is $O\big(g(n)\big)$.
  - $g(n)$ bounds from above
- Big-Omega: Given $f, g: \mathbb{N} \to \mathbb{R}^+$, if there are constants $c > 0$ and $n_0 \geq 1$ such that $f(n) \geq c \cdot g(n)$ for all $n \geq n_0$, then we say that $f(n)$ is $\Omega\big(g(n)\big)$.
  - $g(n)$ bounds from below
- Example: $f(n) = 3n\log n - 2n$ is $\Omega(n \log n)$
  - $3n\log n - 2n = n\log n + 2n(\log n - 1) \geq n\log n$ for $\mathrm{c} = 1, n_0 = 2$

Big-Theta: Big-Oh *and* Big-Omega

- $f(n) = \Theta(g(n))$ if $f(n)$ is both $O(g(n))$ and $\Omega(g(n))$:

$$c' g(n) \leq f(n) \leq c'' g(n), \text{ for } n \geq n_0$$

- Example: $f(n) = 3n \log n + 4n + 5 \log n$ is $\Theta(n \log n)$ since $c' = 1, c'' = 12, n \geq n_0 = 2$ satisfy the above condition.
- We won't be seeing $\Omega(g(n))$ and $\Theta(g(n))$ a lot.

Examples

- $2n = O(n^2)$
- $2n = O(n)$
- $2n^2 = O(n^2)$
- $2n^2 \neq O(n)$

- $2n \neq \Omega(n^2)$
- $2n = \Omega(n)$
- $2n^2 = \Omega(n^2)$
- $2n^2 = \Omega(n^2)$

- $2n \neq \Theta(n^2)$
- $2n = \Theta(n)$
- $2n^2 = \Theta(n^2)$
- $2n^2 \neq \Theta(n)$

$$f(n) \leq c \cdot g(n)$$
$$f(n) \geq c \cdot g(n)$$
$$c' g(n) \leq f(n) \leq c'' g(n)$$

Asymptotic Algorithm Analysis

- Let's asymptotically analyze the worst-case running time with big-Oh
- Better algorithm has a slower growth rate as n grows
- If we want to analyze the running time of an algorithm, then we analyze the time complexity
  - An algorithm which is $O(n)$ has a linear time complexity
  - We may also study the space complexity (how "space" grows as n grows)

n	$\log n$	n	$n \log n$	n^2	n^3	2^n
8	3	8	24	64	512	256
16	4	16	64	256	4,096	65,536
32	5	32	160	1,024	32,768	4,294,967,296
64	6	64	384	4,096	262,144	1.84×10^{19}
128	7	128	896	16,384	2,097,152	3.40×10^{38}
256	8	256	2,048	65,536	16,777,216	1.15×10^{77}
512	9	512	4,608	262,144	134,217,728	1.34×10^{154}

Growth rates of 7 functions

Some Caution

- In practice, the asymptotic growth rates may not make sense
- Sometimes, constants may not be ignored
- $10^{100}n$ is $O(n)$ while $10n\log n$ is $O(n\log n)$
- But we should prefer $10n\log n$ since 10^{100} is just too large!

- Also, some low-order terms may need to be considered
- n^2 vs. $10n^2 + n$
  - They are both $O(n^2)$, but the second one clearly grows faster!
  - When comparing two such algorithms, also consider low-order terms to "find the winner"

Example 1 (Again): `find_max`

- Operations per line: Worst case: $5n + 3$ ops, Best case: $4n + 3$ ops
  - L3: 2 ops
  - L4: $2n$ ops
  - L5: $2n$ ops
  - L6: 0 to n ops
  - L7: 1 op
- We now can say `find_max` on an array of n numbers runs in $O(n)$ time (has linear time complexity)
- Trust your intuition:
  - As n grows, the number of for-loop iteration increases
  - Inside the for-loop, the operations do not depend on n

```
1 def find_max(data):
2   """Return the maximum element from a nonempty Python list."""
3   biggest = data[0]                  # The initial value to beat
4   for val in data:                   # For each value:
5     if val > biggest                 # if it is greater than the best so far,
6       biggest = val                  # we have found a new best (so far)
7   return biggest                     # When loop ends, biggest is the max
```

Example 2: Prefix Averages

- The i-th prefix average of an array X is average of the first $(i+1)$ elements of X:

$$A[i] = (X[0] + X[1] + \cdots + X[i])/(i+1)$$

- What is the time complexity of the code below?

```
def prefix_average1(S):
  """Return list such that, for all j, A[j] equals average of S[0], ..., S[j]."""
  n = len(S)
  A = [0] * n                     # create new list of n zeros
  for j in range(n):
    total = 0                     # begin computing S[0] + ... + S[j]
    for i in range(j + 1):
      total += S[i]
    A[j] = total / (j+1)          # record the average
  return A
```

Example 2: Prefix Averages

- Outer-loop is $O(n)$ and jth inner-loop is $O(j)$
- $\Rightarrow 1 + 2 + \cdots + n = \sum_{i=1}^{n} i = \frac{n(n+1)}{2} = \frac{n^2}{2} + \frac{n}{2}$ operations
- $\Rightarrow O(n^2)$
- Can we make this faster (i.e., slower growth rate)?

```
def prefix_average1(S):
  """Return list such that, for all j, A[j] equals average of S[0], ..., S[j]."""
  n = len(S)
  A = [0] * n                          # create new list of n zeros
  for j in range(n):
    total = 0                          # begin computing S[0] + ... + S[j]
    for i in range(j + 1):
      total += S[i]
    A[j] = total / (j+1)               # record the average
  return A
```

Example 2: Prefix Averages Ver. 2

- Replace the inner loop with the single expression sum(S[0:j+1])
- Asymptotically, this is the same as `prefix_average1`
  - just that sum(S[0:j+1]) is more efficient programming-wise
- Still $O(n^2)$
- Can we make this faster (i.e., slower growth rate)?

```
def prefix_average2(S):
  """Return list such that, for all j, A[j] equals average of S[0], ..., S[j]."""
  n = len(S)
  A = [0] * n                          # create new list of n zeros
  for j in range(n):
    A[j] = sum(S[0:j+1]) / (j+1)       # record the average
  return A
```

Example 2: Prefix Averages Ver. 3

- Outer-loop is $O(n)$ and each iteration is $O(1)$ (constant time)
- $\Rightarrow O(n)$
- `prefix_average3` is a linear time complexity algorithm!

```
def prefix_average3(S):
  """Return list such that, for all j, A[j] equals average of S[0], ..., S[j]."""
  n = len(S)
  A = [0] * n                    # create new list of n zeros
  total = 0                      # compute prefix sum as S[0] + S[1] + ..
  for j in range(n):
    total += S[j]                # update prefix sum to include S[j]
    A[j] = total / (j+1)         # compute average based on current sum
  return A
```

Better Algorithm ⇒ Better Data Structure?

- Are there duplicate elements in the array?
- Intuition: nested for-loop which both depend on n
- $(n-1)+(n-2)+\cdots+2+1 \Rightarrow O(n^2)$
- If we want to perform this uniqueness test (task) on this array (data structure), we expect $O(n^2)$.

```
def unique1(S):
  """Return True if there are no duplicate elements in sequence S."""
  for j in range(len(S)):
    for k in range(j+1, len(S)):
      if S[j] == S[k]:
        return False                    # found duplicate pair
  return True                           # if we reach this, elements were unique
```

Better Algorithm ⇒ Better Data Structure?

- Are there duplicate elements in the array?
- We sort the array first (line 3): best sorting algorithm takes $O(n \log n)$
  - This puts duplicates next to each other
- Then, we just check to see if the element in $j+1$ is the same as the element in j: $O(n)$
- Total time complexity: $O(n \log n) + O(n) \Rightarrow O(n \log n)$
- The same task just got much faster! Better data structure!

```
1 def unique2(S):
2   """Return True if there are no duplicate elements in sequence S."""
3   temp = sorted(S)                    # create a sorted copy of S
4   for j in range(1, len(temp)):
5     if S[j−1] == S[j]:
6       return False                    # found duplicate pair
7   return True                         # if we reach this, elements were unique
```

Summary

- We perform various operations on data structures
  - Add, find, delete, etc.
- Those operations are essentially programs or algorithms
- The complexity of programs/algorithms (time, space, etc.) depend on the problem "size" (input size n)
- Analyzing program/algorithm efficiency allows us to analyze data structures
- We focus on asymptotically analyzing the worst-case time complexity

- Future lectures: As we learn various data structures, we will continuously discuss their "goodness" of data structures by deriving the worst-case time complexity of related operations
  - Sometimes, we can "improve" the data structures with efficient algorithms