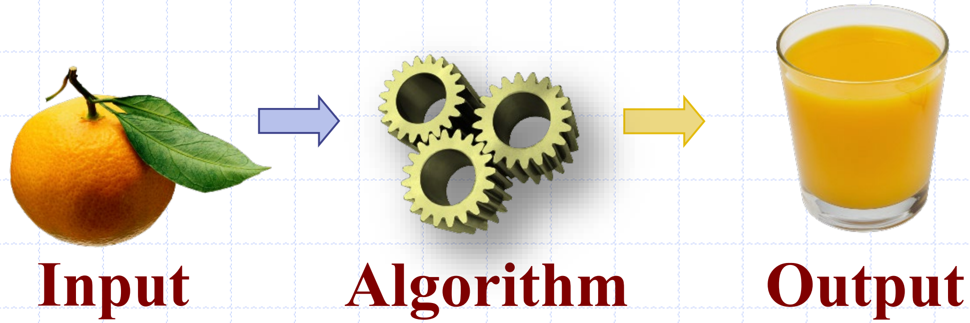
Analysis of Algorithms

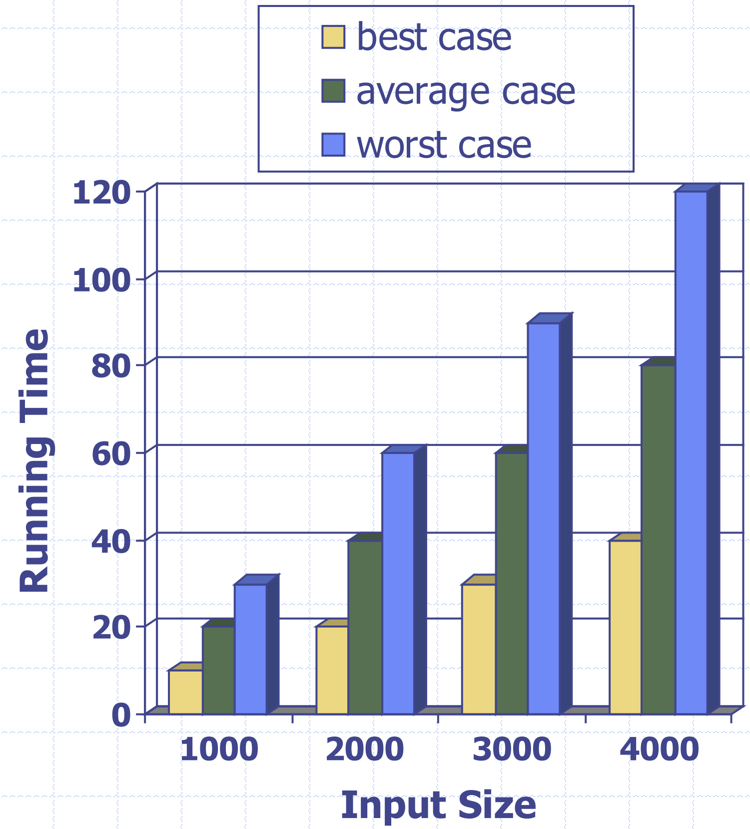


How fast is your algorithm?

- Low memory usage?
- Small amount of time measured on a stopwatch?
- Low power consumption?

Running Time

- Most algorithms transform input objects into output objects.
- The running time of an algorithm typically grows with the input size.
- Average case time is often difficult to determine.
- We focus on the worst case running time.
  - Easier to analyze
  - Crucial to applications such as games, finance and robotics

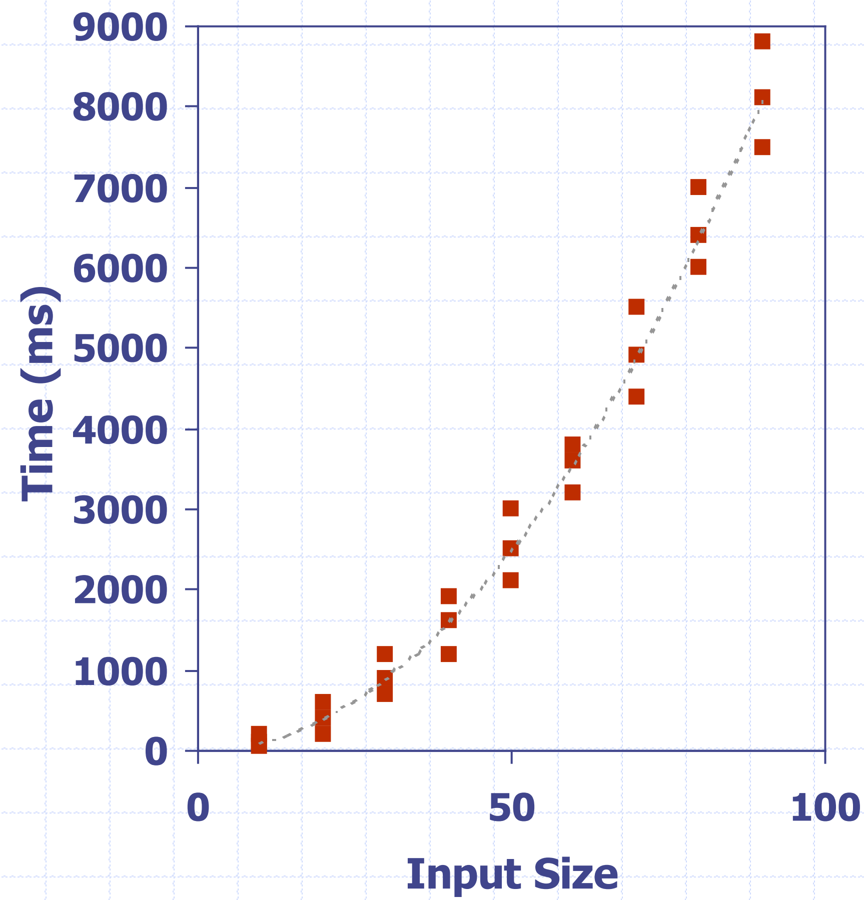


Experimental Studies

- Write a program implementing the algorithm
- Run the program with inputs of varying size and composition, noting the time needed:

```
from time import time
start_time = time( )
run algorithm
end_time = time( )
elapsed = end_time − start_time
```

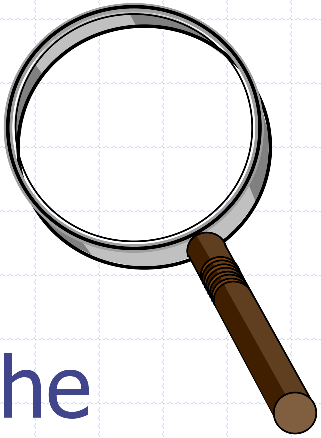
- Plot the results



Limitations of Experiments

- It is necessary to implement the algorithm, which may be difficult
- Results may not be indicative of the running time on other inputs not included in the experiment.
- In order to compare two algorithms, the same hardware and software environments must be used

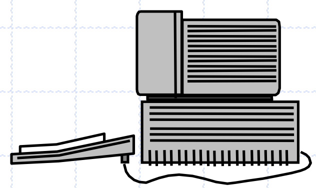
Theoretical Analysis



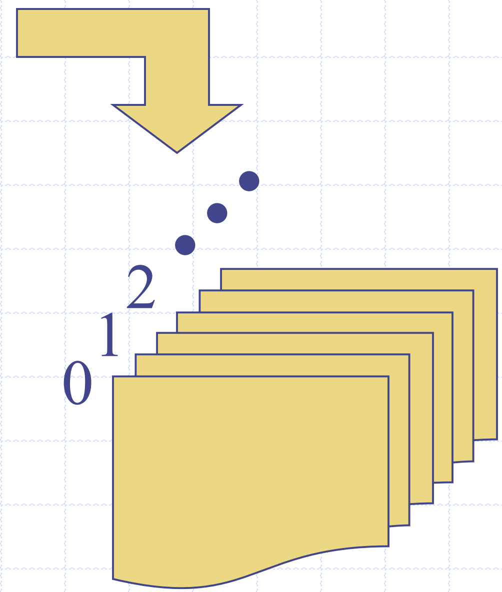
- Uses a high-level description of the algorithm instead of an implementation
- Characterizes running time as a function of the input size, n.
- Takes into account all possible inputs
- Allows us to evaluate the speed of an algorithm independent of the hardware/software environment

The Random Access Machine (RAM) Model

- A **CPU**



- An potentially unbounded bank of **memory** cells, each of which can hold an arbitrary number or character



- Memory cells are numbered and accessing any cell in memory takes unit time.

Elementary Operations

- Algorithmic "time" is measured in elementary operations
  - Math (+, -, *, /)
  - Comparisons (==, >, <=, ...)
  - Function calls and value returns
  - Variable assignment
  - Variable increment or decrement
  - Array allocation
  - Creating a new object (may have elementary ops too!)
- In practice, all of these operations take different amounts of time
- For the purpose of algorithm analysis, we assume each of these operations takes the same time: "1 operation"

Elementary Operations

- Basic computations performed by an algorithm
- Largely independent from the programming language
- Exact definition not important (we will see why later)
- Assumed to take a constant amount of time in the RAM model

- Examples:
  - Evaluating an expression
  - Assigning a value to a variable
  - Indexing into an array
  - Calling a method
  - Returning from a method

Example: Constant Running Time

```
function first(array):
    // Input: an array
    // Output: the first element
    return array[0] // index 0 and return, 2 ops
```

- How many operations are performed in this function if the list has ten elements? If it has 100,000 elements?

Example: Constant Running Time

```
function first(array):
    // Input: an array
    // Output: the first element
    return array[0] // index 0 and return, 2 ops
```

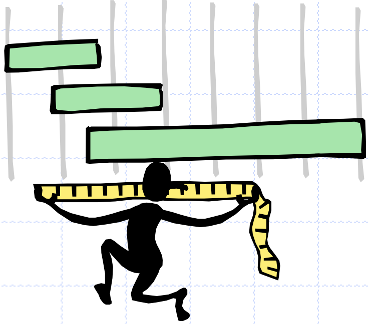
- How many operations are performed in this function if the list has ten elements? If it has 100,000 elements?
  - Always 2 operations performed
  - Does not depend on the input size

Example: Linear Running Time

```
def argmax(array):
   // Input: an array
   // Output: the index of the maximum value
   index = 0 // assignment, 1 op
   for i in range(len(array))://1 + 1 op per loop
      if array[i] > array[index]://3 ops per loop
         index = i // 1 op per loop, sometimes
   return index // 1 op
```

- How many operations if the list has 10 elements? 100,000 elements?
  - Varies proportionally to the size of the input list: $6n + 2$
  - We'll be in the `for` loop longer and longer as the input list grows
  - If we were to plot, the runtime would increase linearly

Estimating Running Time



- Algorithm **argmax** executes $6n + 2$ primitive operations in the worst case, $5n + 2$ in the best case. Define:

  a = Time taken by the fastest primitive operation

  b = Time taken by the slowest primitive operation

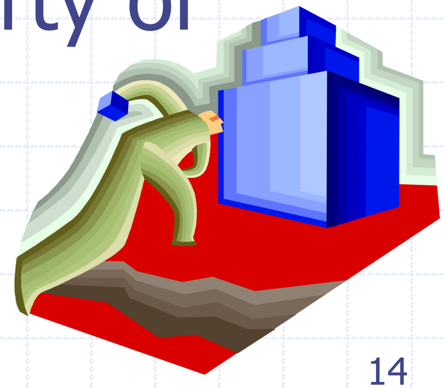
- Let $T(n)$ be worst-case time of **argmax.** Then

$$a\,(5n + 2) \leq T(n) \leq b(6n + 2)$$

- Hence, the running time $T(n)$ is bounded by two linear functions.

Growth Rate of Running Time

- Changing the hardware/ software environment
  - Affects $T(n)$ by a constant factor, but
  - Does not alter the growth rate of $T(n)$
- The linear growth rate of the running time $T(n)$ is an intrinsic property of algorithm **argmax**



Example: Quadratic Running Time

```
def possible_products(array):
    // Input: an array
    // Output: a list of all possible products
    //    between any two elements in the list
    products = [] // make an empty list, 1 op
    for i in range(len(array)): // 1+1=2 op per loop
        for j in range(len(array)): // 1+1=2 op per loop per loop
            products.append(array[i] * array[j]) // 4 ops per loop per loop
    return products // 1 op
```

- Requires about $6n^2 + 2n + 2$ operations (okay to approximate!)
  - If we were to plot this, the number of operations executed grows quadratically!
- Consider adding one element to the list: the added element must be multiplied with every other element in the list
- Notice that the linear algorithm on the slide #14 had only one `for` loop, while this quadratic one has `two` for loops, nested. What would be the highest-degree term (in number of operations) if there were three nested loops?

Some Common Computing Times

$\log_2 n$	n	$n \log_2 n$	n^2	2^n
1	2	2	4	4
2	4	8	16	16
3	8	24	64	256
4	16	64	256	65,536
5	32	160	1,024	4,294,967,296
6	64	384	4,096	1.84×10^{19}
7	128	896	16,384	3.40×10^{38}
8	256	2,048	65,536	1.16×10^{77}
9	512	4,608	262,144	1.34×10^{154}
10	1,024	10,240	1,048,576	1.80×10^{308}

Slide by Matt Stallmann included with permission.

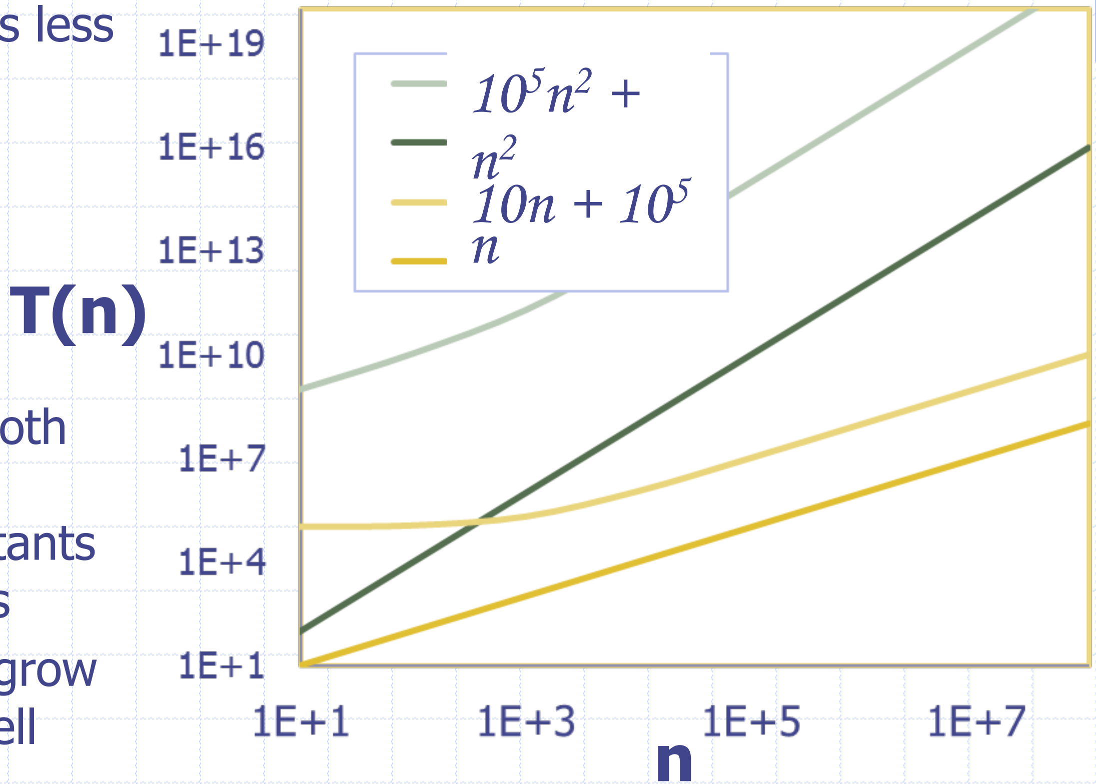
Why Growth Rate Matters

if runtime is...	time for n + 1	time for 2 n	time for 4 n
$c \lg n$	$c \lg (n + 1)$	$c (\lg n + 1)$	$c(\lg n + 2)$
$c n$	$c (n + 1)$	$2c n$	$4c n$
$c n \lg n$	$\sim c n \lg n + c n$	$2c n \lg n + 2cn$	$4c n \lg n + 4cn$
$c n^2$	$\sim c n^2 + 2c n$	$\mathbf{4c n^2}$	$16c n^2$
$c n^3$	$\sim c n^3 + 3c n^2$	$8c n^3$	$64c n^3$
$c 2^n$	$c 2^{n+1}$	$c 2^{2n}$	$c 2^{4n}$

runtime quadruples when problem size doubles

Summarizing Function Growth

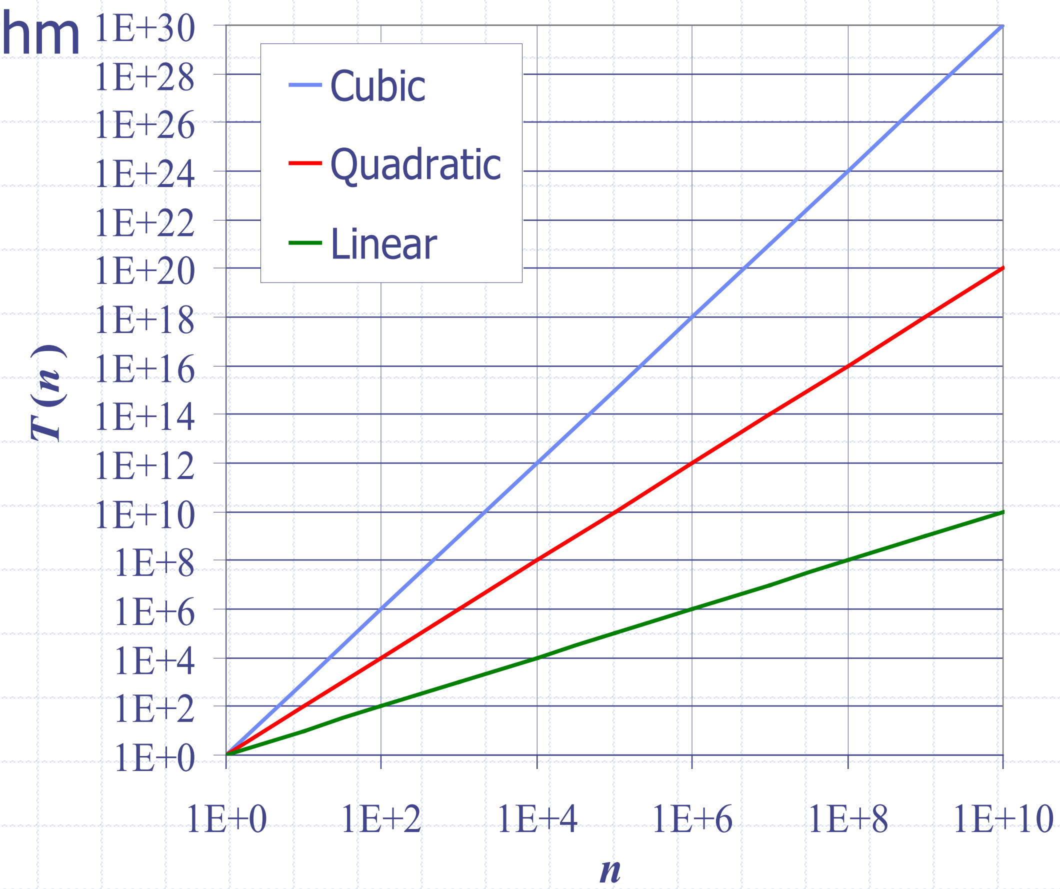
- For very large inputs, the growth rate of a function becomes less affected by:
  - constant factors or
  - lower-order terms
- Examples
  - $10^5n^2 + 10^8n$ and n^2 both grow with same slope despite differing constants and lower-order terms
  - $10n + 10^5$ and n both grow with same slope as well



In this graph (log scale on both axes), the slope of a line corresponds to the growth rate of its respective function

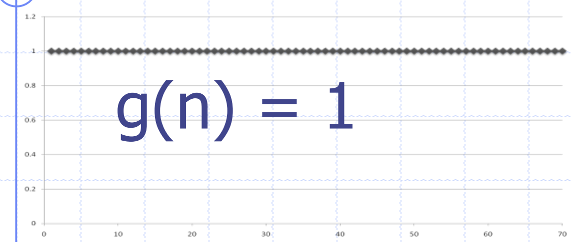
Seven Important Functions

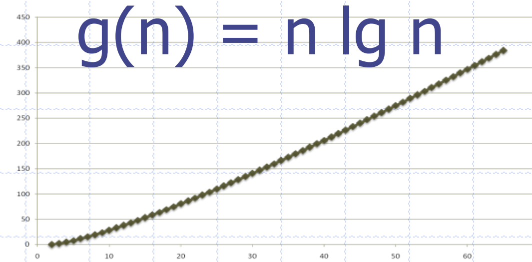
- Seven functions that often appear in algorithm analysis:
  - Constant ≈ 1
  - Logarithmic ≈ $\log n$
  - Linear ≈ n
  - N-Log-N ≈ $n \log n$
  - Quadratic ≈ n^2
  - Cubic ≈ n^3
  - Exponential ≈ 2^n
- In a log-log chart, the slope of the line corresponds to the growth rate

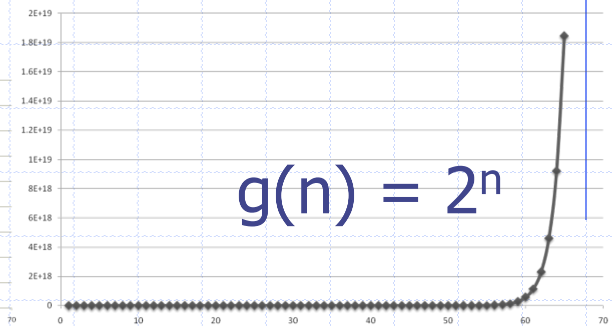


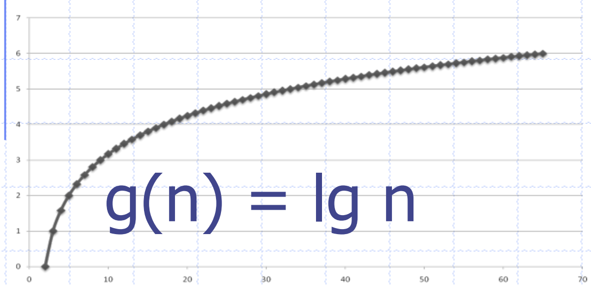
Functions Graphed Using “Normal” Scale

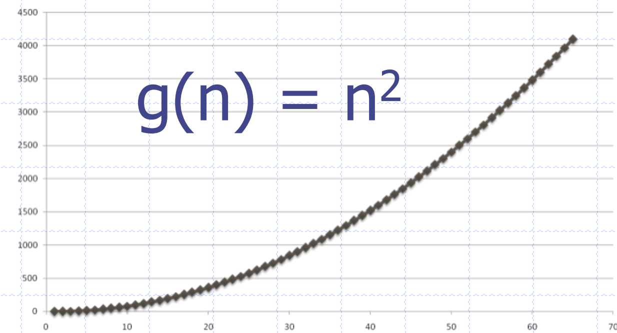
Slide by Matt Stallmann included with permission.

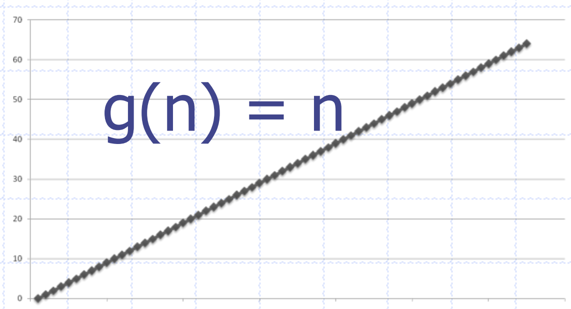


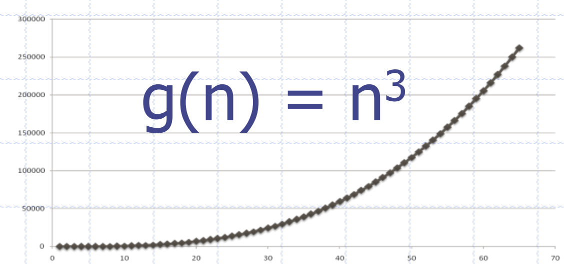




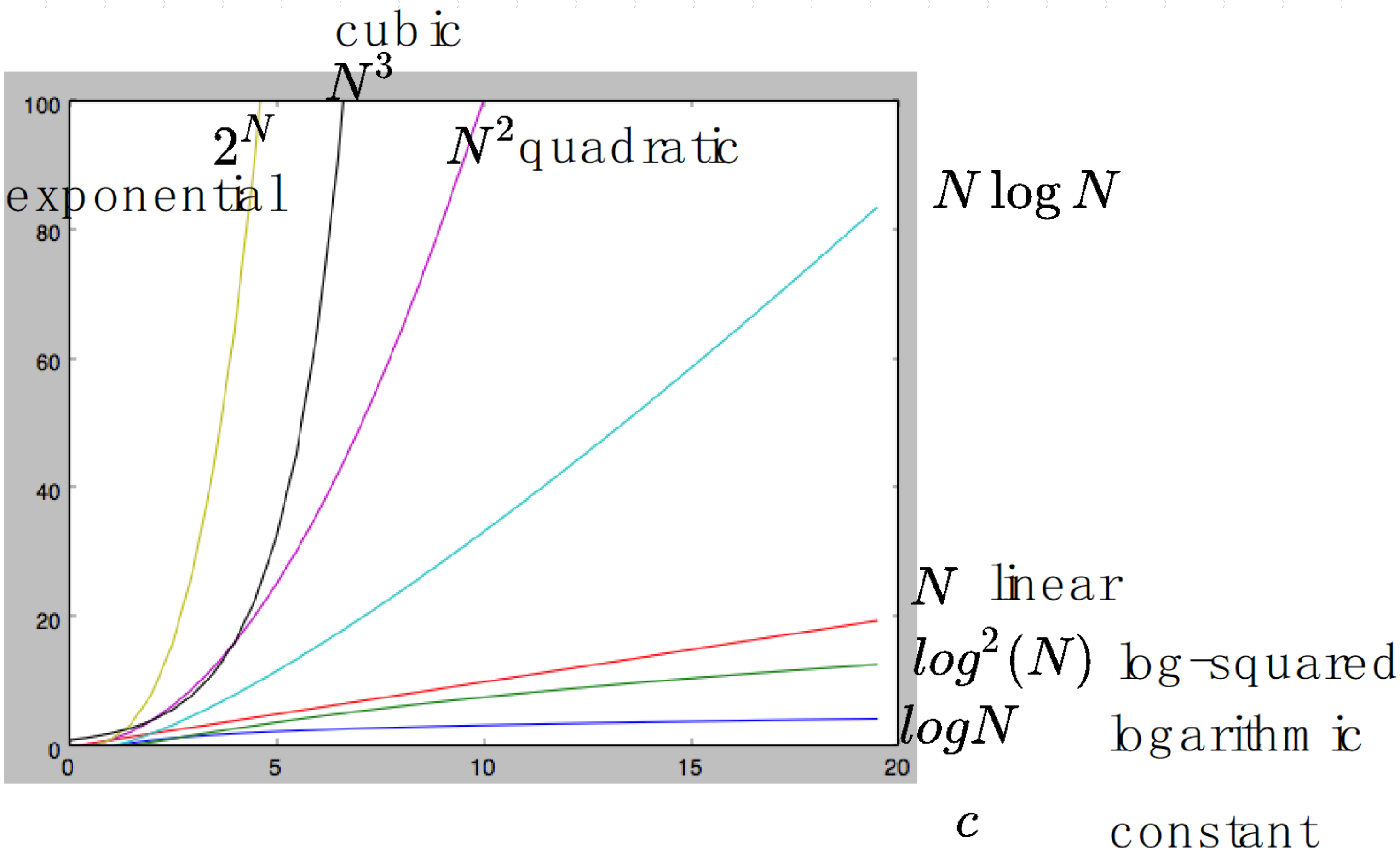






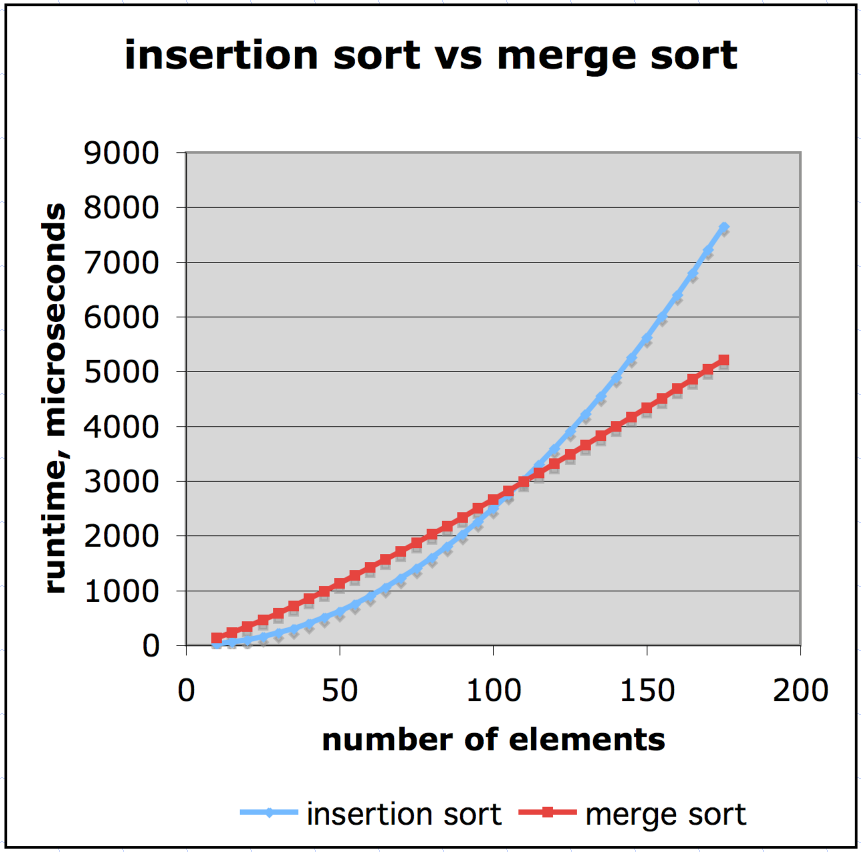


Typical Growth Rates



Slide by Matt Stallmann included with permission.

Comparison of Two Algorithms



insertion sort is

$n^2 / 4$

merge sort is

2 n lg n

sort a million items?

insertion sort takes roughly 70 hours

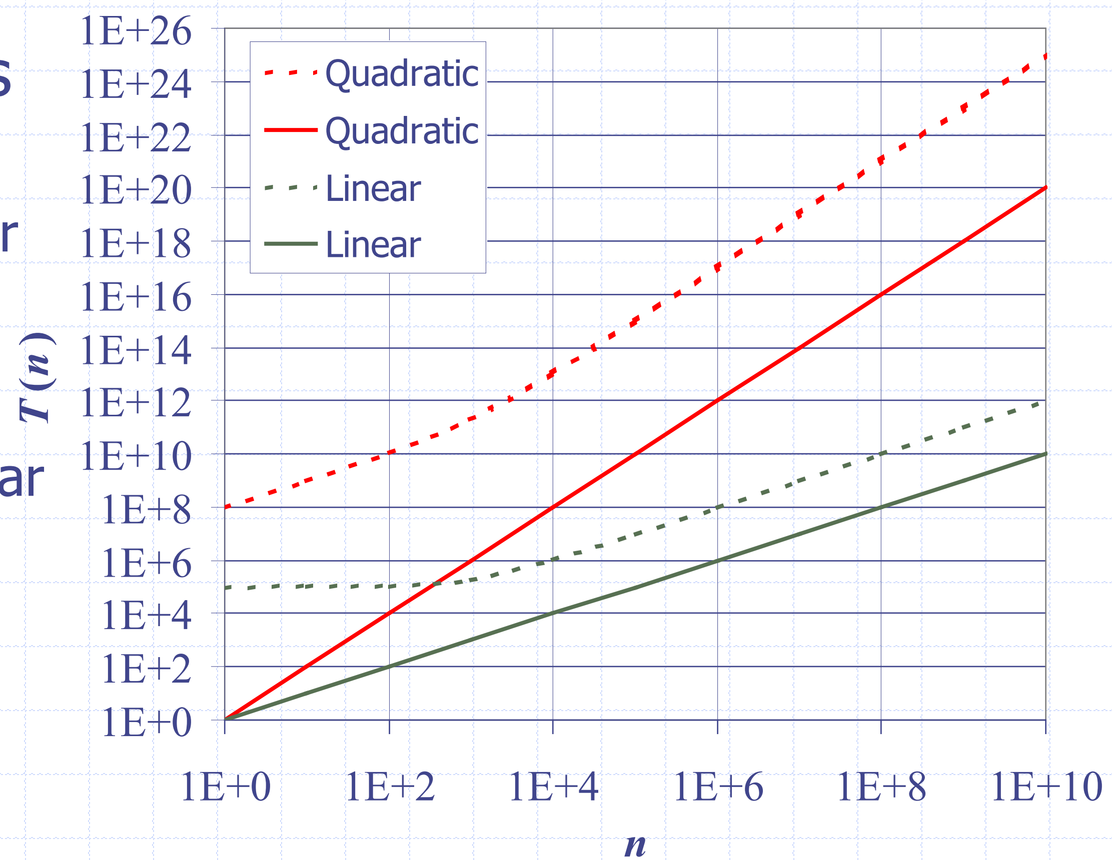
while

merge sort takes roughly 40 seconds

This is a slow machine, but if 100 x as fast then it's 40 minutes versus less than 0.5 seconds

Constant Factors

- The growth rate is not affected by
  - constant factors or
  - lower-order terms
- Examples
  - $10^2 n + 10^5$ is a linear function
  - $10^5 n^2 + 10^8 n$ is a quadratic function



Comparison of Insertion Sort and Python Built In Sort Function

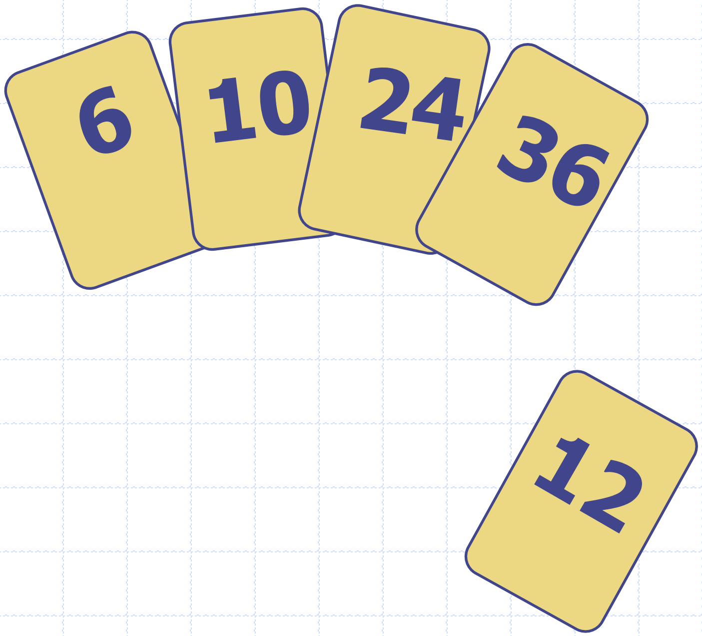
- Please go to Brightspace to open the InsertionVSbuiltinClassVersion.py file
- Implement Insertion sort in that code.
- Use the Python built in sort from list class
- Compare the runtime. Which one is better???
- Submit code to Gradescope

Insertion Sort

- Idea: like sorting a hand of playing cards
  - Start with an empty left hand and the cards facing down on the table.
  - Remove one card at a time from the table, and insert it into the correct position in the left hand
    - compare it with each of the cards already in the hand, from right to left
  - The cards held in the left hand are sorted
    - these cards were originally the top cards of the pile on the table

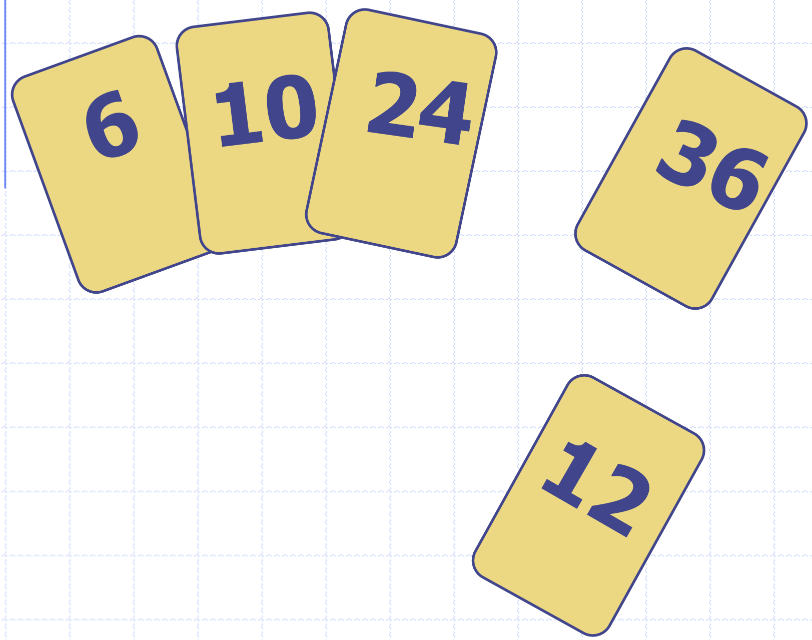
Visualization: https://visualgo.net/en/sorting

Insertion Sort

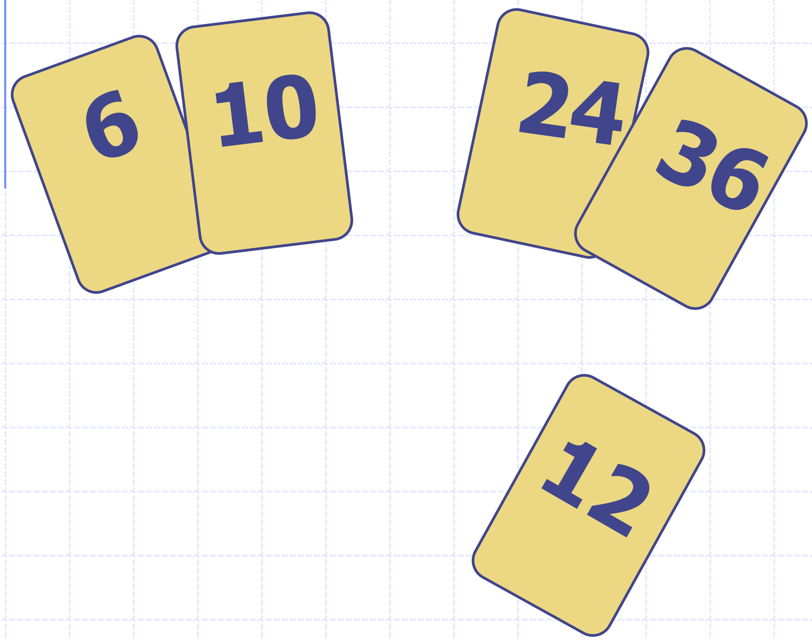


To insert 12, we need to make room for it by moving first 36 and then 24.

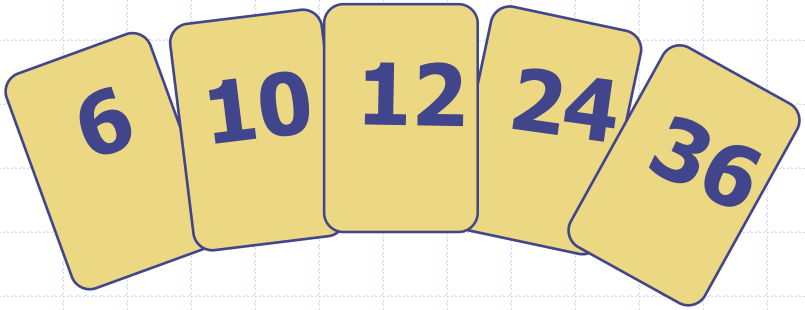
Insertion Sort



Insertion Sort



Insertion Sort



Insertion Sort

input array

5 2 4 6 1 3

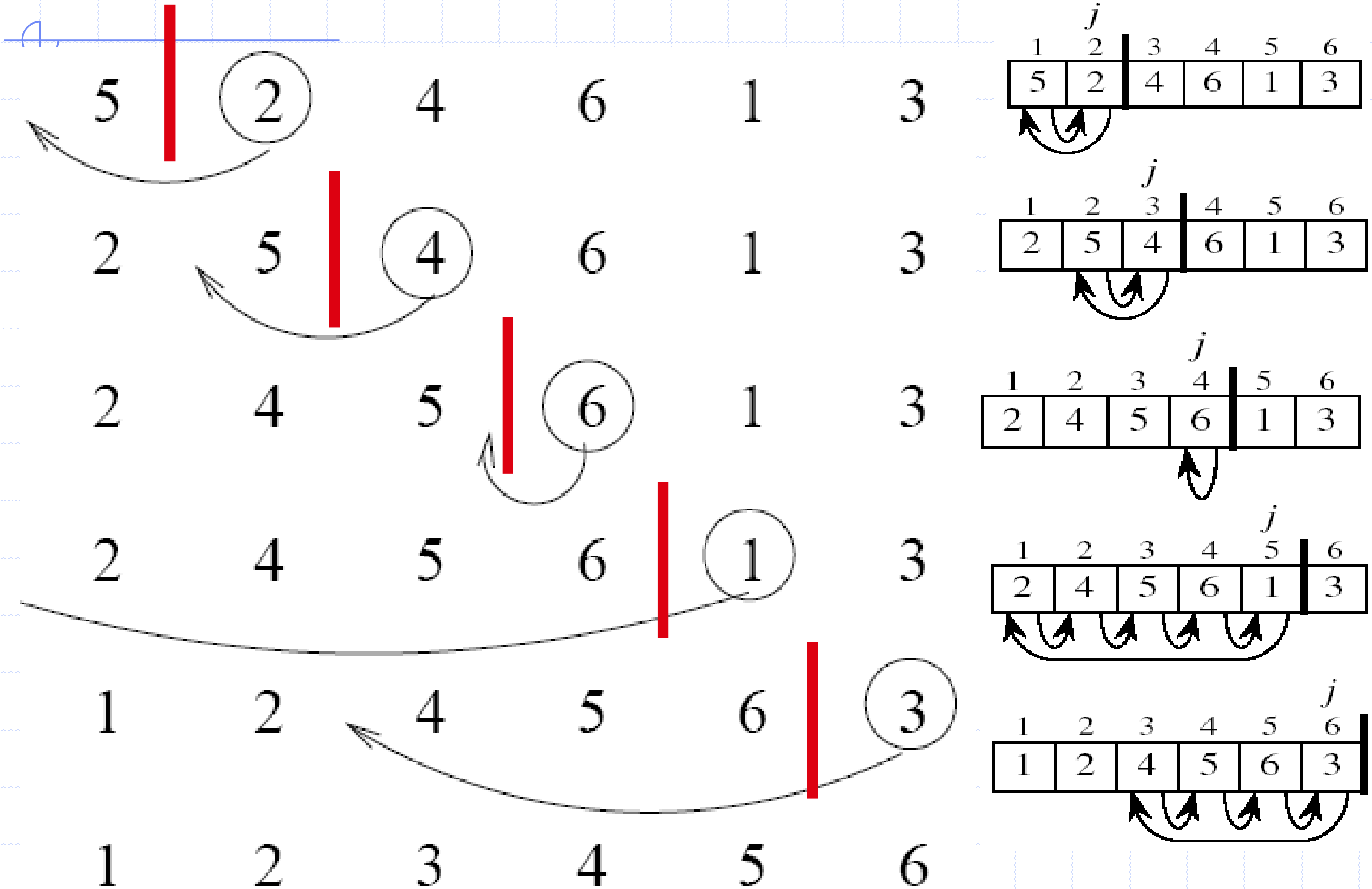
at each iteration, the array is divided in two sub-arrays:

left sub-array right sub-array

2 5 4 6 1 3

sorted unsorted

Insertion Sort



INSERTION-SORT

0	1	2	3	4	5	6	7
a_0	a_1	a_2	a_3	a_4	a_5	a_6	a_7

key

Alg.: INSERTION-SORT*(A)*

```
for j ← 1 to n-1
      do key ← A[ j ]
▷         # Insert key into the sorted sequence A[0 . . j-1]
          i ← j - 1
          while i >= 0 and A[i] > key
                do A[i + 1] ← A[i]
                   i ← i - 1
          A[i + 1] ← key
```

- Insertion sort – sorts the elements in place

Big-O Notation

- Given functions $f(n)$ and $g(n)$, we say that

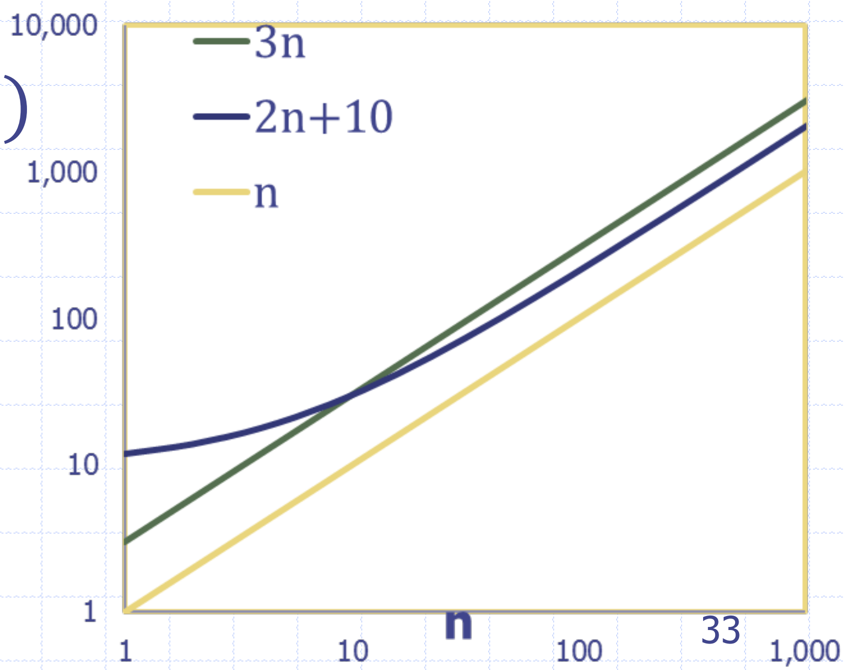
$$f(n) \text{ is } O(g(n))$$

if there exist positive constants c and n_0 such that

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

- Example: $2n + 10$ is $O(n)$
  - Pick $c = 3$ and $n_0 = 10$

$$2n + 10 \leq 3n$$
$$2(10) + 10 \leq 3(10)$$
$$30 \leq 30$$



Big-O Notation (continued)

Example: n^2 is not $O(n)$

$$n^2 \leq cn$$

$$n \leq c$$

The above inequality cannot be satisfied because c must be a constant, therefore for any $n > c$ the inequality is false

Big-O and Growth Rate

- Big-O notation gives an upper bound on the growth rate of a function
- We say "an algorithm is $O(g(n))$" if the growth rate of the algorithm is no more than the growth rate of $g(n)$
- We saw on the previous slide that n^2 is not $O(n)$
  - But n is $O(n^2)$
  - And n^2 is $O(n^3)$
  - Why? Because Big-O is an upper bound!

Summary of Big-O Rules

- If $f(n)$ is a polynomial of degree d, then $f(n)$ is $O(n^d)$. In other words:
  - forget about lower-order terms
  - forget about constant factors
- Use the smallest possible degree
  - It's true that $2n$ is $O(n^{50})$, but that's not a helpful upper bound
  - Instead, say it's $O(n)$, discarding the constant factor and using the smallest possible degree

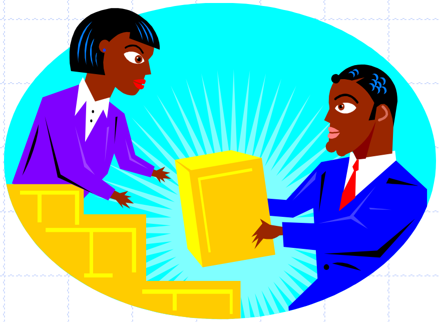
Constants in Algorithm Analysis

- Find the number of primitive operations executed as a function (T) of the input size
  - `first`: $T(n) = 2$
  - `argmax`: $T(n) = 6n + 2$
  - `possible_products`: $T(n) = 6n^2 + 2n + 2$
- In the future we can skip counting operations and replace any constants with c since they become irrelevant as n grows
  - `first`: $T(n) = c$
  - `argmax`: $T(n) = c_0 n + c_1$
  - `possible_products`: $T(n) = c_0 n^2 + n + c_1$

Big-O in Algorithm Analysis

- Easy to express T in big-O by dropping constants and lower-order terms
- In big-O notation
  - `first` is $O(1)$
  - `argmax` is $O(n)$
  - `possible_products` is $O(n^2)$
- The convention for representing $T(n) = c$ in big-O is $O(1)$.

More Big-Oh Examples



- $7n-2$

  $7n-2$ is $O(n)$

  need $c > 0$ and $n_0 \geq 1$ such that $7n-2 \leq c \bullet n$ for $n \geq n_0$

  this is true for $c = 7$ and $n_0 = 1$

- $3n^3 + 20n^2 + 5$

  $3n^3 + 20n^2 + 5$ is $O(n^3)$

  need $c > 0$ and $n_0 \geq 1$ such that $3n^3 + 20n^2 + 5 \leq c \bullet n^3$ for $n \geq n_0$

  this is true for $c = 4$ and $n_0 = 21$

- $3 \log n + 5$

  $3 \log n + 5$ is $O(\log n)$

  need $c > 0$ and $n_0 \geq 1$ such that $3 \log n + 5 \leq c \bullet \log n$ for $n \geq n_0$

  this is true for $c = 8$ and $n_0 = 2$

Big-Oh and Growth Rate

- The big-Oh notation gives an upper bound on the growth rate of a function
- The statement "$f(n)$ is $O(g(n))$" means that the growth rate of $f(n)$ is no more than the growth rate of $g(n)$
- We can use the big-Oh notation to rank functions according to their growth rate

	$f(n)$ is $O(g(n))$	$g(n)$ is $O(f(n))$
$g(n)$ grows more	Yes	No
$f(n)$ grows more	No	Yes
Same growth	Yes	Yes

Asymptotic Algorithm Analysis

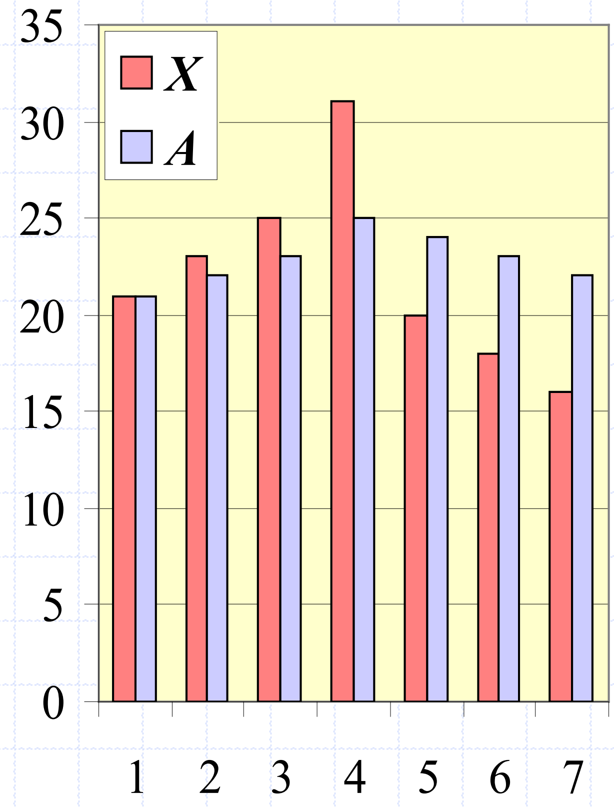
- The asymptotic analysis of an algorithm determines the running time in big-Oh notation
- To perform the asymptotic analysis
  - We find the worst-case number of primitive operations executed as a function of the input size
  - We express this function with big-Oh notation
- Example:
  - We say that algorithm **argmax** "runs in $O(n)$ time"
- Since constant factors and lower-order terms are eventually dropped anyhow, we can disregard them when counting primitive operations

Computing Prefix Averages

- We further illustrate asymptotic analysis with two algorithms for prefix averages
- The i-th prefix average of an array X is average of the first $(i + 1)$ elements of X:

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

- Computing the array A of prefix averages of another array X has applications to financial analysis



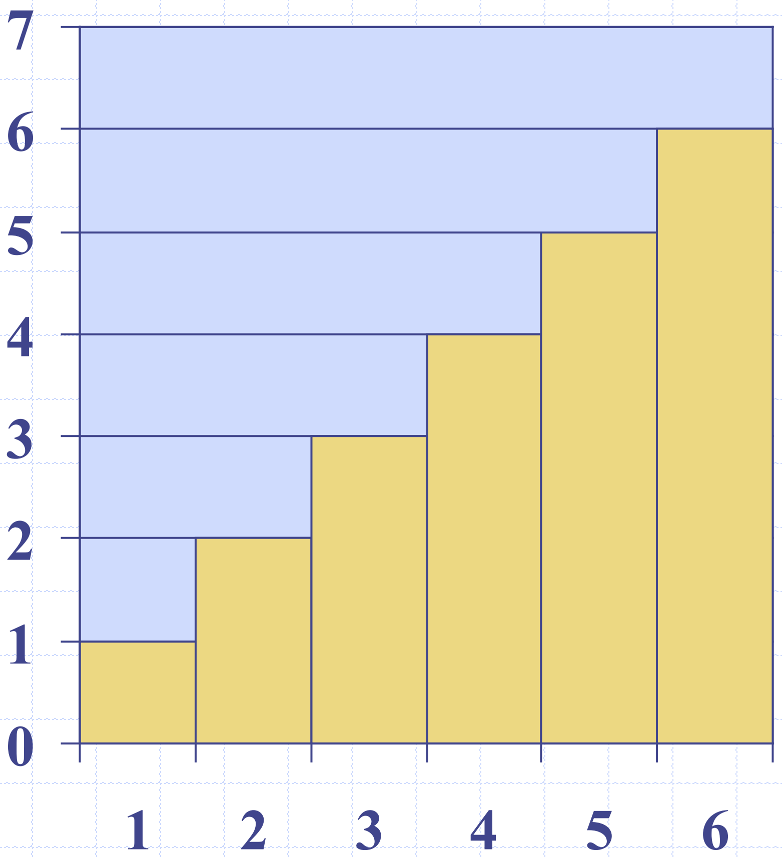
Prefix Averages (Quadratic)

- The following algorithm computes prefix averages in quadratic time by applying the definition

```
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
```

Arithmetic Progression

- The running time of ***prefixAverage1*** is $O(1 + 2 + \ldots + n)$
- The sum of the first n integers is $n(n + 1) / 2$
  - There is a simple visual proof of this fact
- Thus, algorithm ***prefixAverage1*** runs in $O(n^2)$ time



Prefix Averages 2 (Looks Better)

- The following algorithm uses an internal Python function to simplify the code

```
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
```

- Algorithm *prefixAverage2* **still** runs in $O(n^2)$ time!

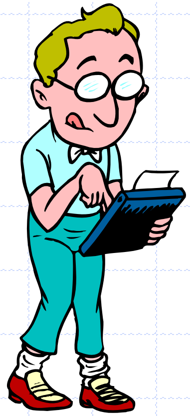
Prefix Averages 3 (Linear Time)

- The following algorithm computes prefix averages in linear time by keeping a running sum

```
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
```

- Algorithm ***prefixAverage3*** runs in $O(n)$ time

Math you need to Review



- Summations
- Logarithms and Exponents
  - **properties of logarithms:**
    - $\log_b(xy) = \log_b x + \log_b y$
    - $\log_b (x/y) = \log_b x - \log_b y$
    - $\log_b x^a = a\log_b x$
    - $\log_b a = \log_x a/\log_x b$
  - **properties of exponentials:**
    - $a^{(b+c)} = a^b a^c$
    - $a^{bc} = (a^b)^c$
    - $a^b / a^c = a^{(b-c)}$
    - $b = a^{\log_a b}$
    - $b^c = a^{c*\log_a b}$
- Proof techniques
- Basic probability

Composition Rules for Big-O

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

$T_1(N) + T_2(N) \quad = \quad O(f(N)) + O(g(N))$

$O(max(f(N), g(N))$

$T_1(N) * T_2(N) \quad = \quad O(f(N)) * O(g(N))$

General Rules – Basic for-loops

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

```java
public static int sum(int n){
  int partialSum = 0;

  for (int i = 1; i <= n; i++)
        partialSum += i * i * i;
  return partialSum;
}
```

1 step (initialization) +1 step for last test

`int partialSum = 0;`: 1 step

N iterations

`for (int i = 1; i <= n; i++)`: 2 steps each

`partialSum += i * i * i;`: 4 steps each

`return partialSum;`: 1 step

```python
def sum(n):
   partialSum = 0
   for i in range(1,n+1):
      partialSum += i*i*i
   return partialSum
```

T(N) = 6N + 2 = O(N)

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

If loop runs a constant number of times: *O(c)*

General Rules – Nested Loops

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

N iterations $O(N) * O(N) = O(N^2)$

N iterations $O(N)$

2 steps each $O(c)$

```
for i in range(n):
    for j in range(n):
        k += 1
```

General Rules – Consecutive Blocks

```
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;
```

$O(N)$

$O(N^2)$

```
for i in range(n):
  a[i] = 0
for i in range(n):
  for j in range(n):
    a[i] += a[j] + i + j
```

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

General Rules - Conditionals

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

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

$T_{S_1}(N)$

$T_{S_2}(N)$

```
if condition:
    S1
else:
    S2
```

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

Logarithms in the Runtime

```java
public static int binarySearch(int[] a, int x) {
  int low = 0;
  int  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; // Not found.
}
```

```python
def binarySearch(a, x):
    low = 0
    high = len(a) – 1

    while low <= high:
        mid = (low+high) // 2
        if a[mid] < x:
            low = mid+1
        elif a[mid] > x:
            high = mid-1
        else:
            return mid #found
    return -1 #not found
```

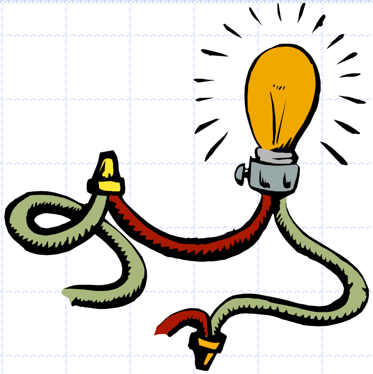
Reduces the search space by half at every step

k steps until $N+1 \geq 2^k \geq N$

$$\mathrm{Log}_2(N+1) \geq k \geq \mathrm{Log}_2 N$$

$T(N) = O(\mathrm{Log}(N))$

In-class exercise



- Please complete the in-class Big-O exercises on Gradescope.

Big-Omega (Ω)

- Recall that $f(n)$ is $O(g(n))$ if $f(n) \leq cg(n)$ for some constant c as n grows
  - Big-O expresses the idea that $f(n)$ grows no faster than $g(n)$
  - $g(n)$ acts as an upper bound to $f(n)$'s growth rate

- What if we want to express a lower bound?

Big-Omega

- We say $f(n)$ is $\Omega(g(n))$ if $f(n) \geq cg(n)$
  - $f(n)$ grows no **slower** than $g(n)$

Big-Theta (Θ)

- What about an upper *and* lower bound?

Big-Theta

- We say f(n) is $\Theta(g(n))$ if

  f(n) is $O(g(n))$ **and** $\Omega(g(n))$

  - f(n) grows the same as g(n) (tight-bound)

Some More Examples

Function, f(n)	Big-Θ
$an + b$	$\Theta(n)$
$an^2 + bn + c$	$\Theta(n^2)$
a	$\Theta(1)$
$3^n + an^{40}$	$\Theta(3^n)$
$an + b \log n$	$\Theta(n)$

Common Time Complexities

Name	Running Time
Constant	$O(1)$
Log-logarithmic	$O(\log \log N)$
Logarithmic	$O(\log N)$
Polylogarithmic	$O((\log N)^2)$
Fractional power	$O(N^c)$ where $0 < c < 1$
Linear	$O(N)$
Linearithmic	$O(N \log N)$
Quadratic	$O(N^2)$
Cubic	$O(N^3)$
Polynomial	$O(N^c)$ where $c > 3$
Exponential	$O(c^N)$ where $c \geq 2$
Factorial	$O(N!)$

source: https://en.wikipedia.org/wiki/Time_complexity#Table_of_common_time_complexities

Relatives of Big-Oh

- **big-Omega**
  - f(n) is $\Omega(g(n))$ if there is a constant $c > 0$ and an integer constant $n_0 \geq 1$ such that $f(n) \geq c \bullet g(n)$ for $n \geq n_0$
- **big-Theta**
  - f(n) is $\Theta(g(n))$ if there are constants $c' > 0$ and $c'' > 0$ and an integer constant $n_0 \geq 1$ such that $c' \bullet g(n) \leq f(n) \leq c'' \bullet g(n)$ for $n \geq n_0$