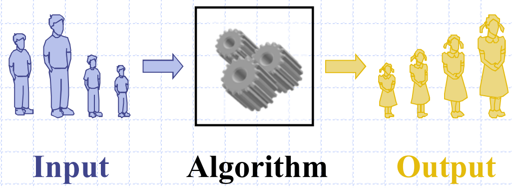
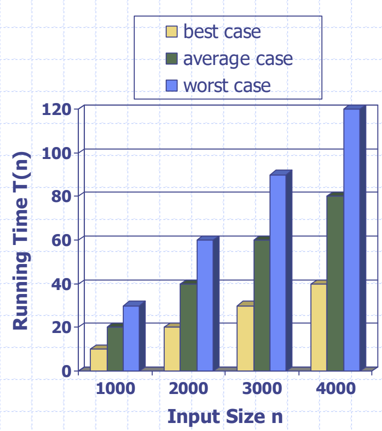
Analysis of Algorithms



An **algorithm** is a step-by-step procedure for solving a problem in a finite amount of time.

Running Time

- Most algorithms transform input objects (data) into output objects (data).
- The running time of an algorithm typically grows with the input size.
- Average case time is often difficult to determine.
  - Need to have extra information about the input.
- We mostly focus on the worst case running time.
  - Easier to analyze

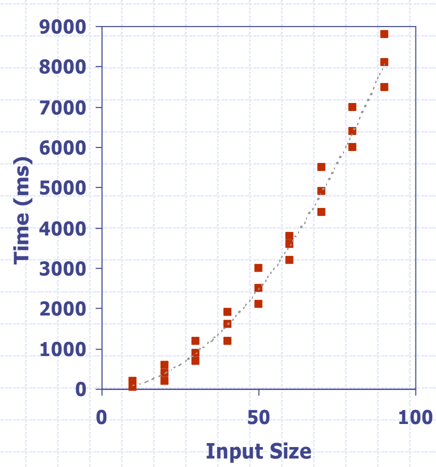


Running Time

- The running time of an algorithm depends on
  - Computer - quality and type
  - Programmer skill
  - Input - types and amounts
  - Algorithm
    - Different ways to solve the same problem - choose wisely

Experimental Studies

- Write a program implementing the algorithm
- Run the program with inputs of varying size and composition
- Measure the runtime using *time*
- Plot the results



Limitations of Experiments

- It is necessary to implement the algorithm first, which may be difficult
- Good range of inputs?
  - Real world data
  - Synthetically created data
- In order to compare two algorithms, the same hardware and software environments must be used
- Difficult to be exhaustive, or use enough sample inputs to be able to make reliable claims about the algorithm.
  - There can be some input that completely bogs down your algorithm, but was never tested

Theoretical Analysis - Basics

- Uses a high-level description (pseudo code) of the algorithm instead of an actual implementation
- Characterizes running time as a function of the input size, *n*.
  - We care about very large input sizes, large n
- Takes into account all possible inputs
- Evaluates algorithm independent of hardware, implementation, input set, etc.
- Count operations not actual clock time

Theoretical Analysis - Benefits

- We want to be able to determine how efficient an algorithm is, across all machines, languages, etc.
- Allows us to compare different approaches to the same problem, i.e. "sorting".
- Helps us identify sections of algorithm with high cost where:
  - Can improve.
  - Cannot improve, *i.e.* lower-bound.
- Asymptotic Analysis – Compare running time as a function of the input size in the limit *i.e.* as *n* approaches infinity.

Pseudocode

- High-level description of an algorithm
- More structured than English prose
- Less detailed than a program
- Preferred notation for describing algorithms
- Hides program design issues

Example: find max element of an array

Algorithm *arrayMax*(A, n)
  Input array A of n integers
  Output maximum element of A

  $currentMax \leftarrow A[0]$
  for $i \leftarrow 1$ **to** $n - 1$ **do**
    if $A[i] > currentMax$ **then**
      $currentMax \leftarrow A[i]$
  return $currentMax$

Pseudocode & C++

Algorithm ***arrayMax***(A,***n***)
  Input: *An array A storing n ≥ 1 integers.*
  Output: *The maximum element of A.*
  currentMax ← *A[0]*
  for i ← n **to** n − 1 **do**
    if ***currentMax*** < *A[i]* **then**
      currentMax ← *A[i]*
  return ***currentMax***

```
int arrayMax(int A[], int n){
    int currentMax = A[0];
    for(int i = 1; i < n; ++i){
        if(currentMax < A[i]){
            currentMax = A[i];
        }
        return currentMax;
    }
}
```


Counting Primitive Operations

- Basic computations performed by an algorithm
  - Assigning a value to a variable
  - Calling a function
  - Performing an arithmetic operation, i.e. 5+7
  - Comparing two numbers
  - Indexing into an array
  - Evaluating an expression
  - Returning from a function
- Primitive Operation-
  - a low level instruction whose execution time depends on environment's hardware and software
  - For analysis purposes, constant time instruction, O(1)

Counting Primitive Operations

- By inspecting the pseudocode, we can determine the maximum number of primitive operations executed by an algorithm, as a function of the input size

Algorithm *printArray*(A, n)

$i \leftarrow 0$	1 assignment
while $i <$ n **do**	n + 1 comparisons
cout << A[i] << endl	n outputs
i ++	n increments

1 + (n+1) + n + n = 3n + 2 operations
Proportional to n, more items = more time

Counting Primitive Operations

- In class exercise

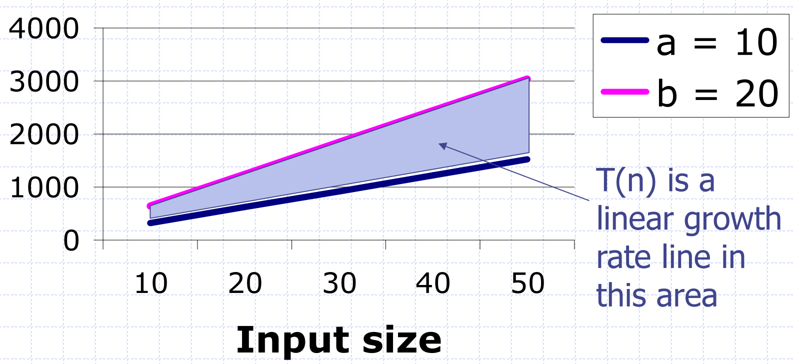
Algorithm *foo*(n)

$x \leftarrow 0$, $y \leftarrow 0$
while $x <$ n **do**
  x ++
  while $y <$ n **do**
    y ++
  $y = 0$

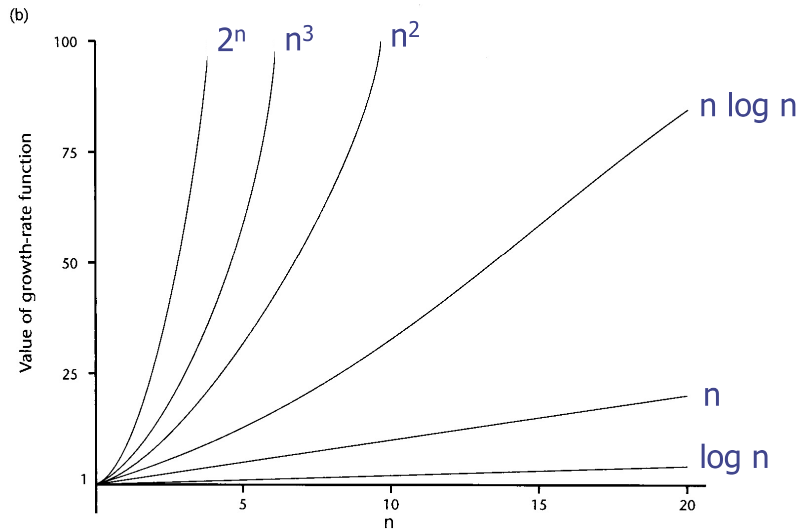
Estimating Running Time

- Algorithm ***printArray*** executes $3n + 2$ primitive operations in the worst 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 ***printArray.*** Then
  $$a\,(3n + 2) \le T(n) \le b(3n + 2)$$
- Hence, the running time $T(n)$ is bounded by two linear functions

Growth Rate of Running Time



Growth Rates

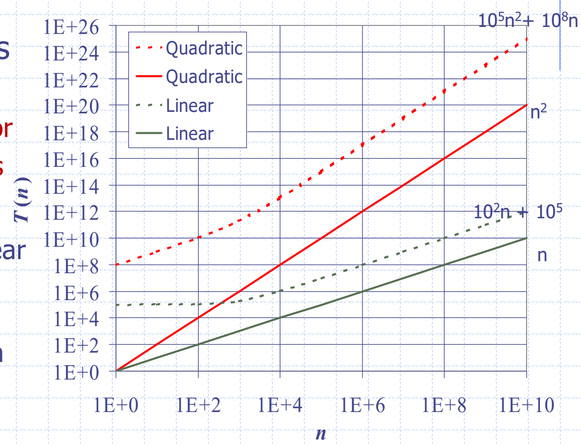


Asymptotic Notation

- Special Classes of Algorithms
  - Constant : *O(1)*
  - Logarithmic *O(log n)*
  - Linear: *O(n)*
  - Logarithmic: *O(n log n)*
  - Quadratic: $O(n^2)$
  - Cubic: $O(n^3)$
  - Exponential: $O(2^n)$

Constant Factors and Low-Order Terms

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



Constant Factors and Low-Order Terms

- Examples
  - ***2n*** and ***100n*** have the same relative growth rates
  - ***10n*** and ***10n + 4*** have the same relative growth rates
  - ***$3n^2$ + 10n + 7*** and ***n^2*** have the same relative growth rates
  - ***10000n + 1000*** and ***n*** have the same relative growth rates

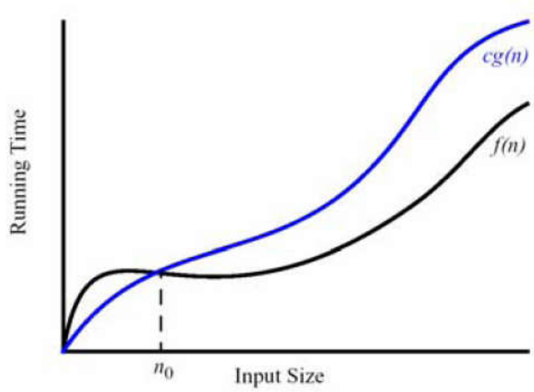
Big-Oh Notation

- Given functions $f(n)$ and $g(n)$, we say that $f(n)$ is $O(g(n))$ if there are positive constants c and n_0 such that:

$$f(n) \leq cg(n) \text{ for } n \geq n_0$$

- We say…
  - "$f(n)$ is ***Big-Oh*** of $g(n)$" or
  - "$f(n)$ is ***order*** $g(n)$"

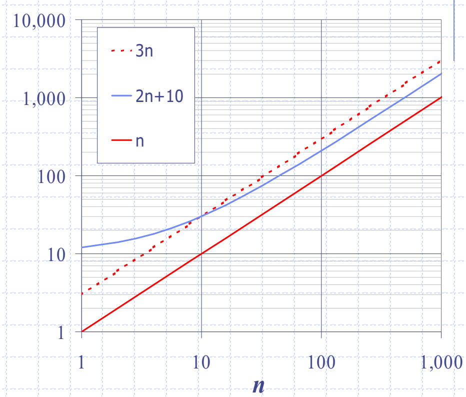
"Big-Oh" Illustrated



The function $f(n)$ is $O(g(n))$, for $f(n) \leq c \cdot g(n)$ when $n \geq n_0$

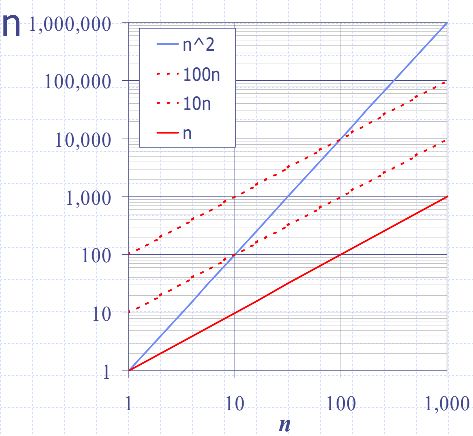
Big-Oh Notation

- ***Definition*** - Given functions $f(n)$ and $g(n)$, we say that $f(n)$ is $O(g(n))$ if there are positive constants c and n_0 such that
  $f(n) \le cg(n)$ for $n \ge n_0$
- Example: $2n + 10$ is $O(n)$
  - $2n + 10 \le cn$
  - $10 \le (c - 2)\, n$
  - $10/(c - 2) \le n$
  - Pick $c = 3$ and $n_0 = 10$



Big-Oh Example

- Example: the function n^2 is NOT $O(n)$
  - $n^2 \le cn$
  - The above inequality cannot be satisfied since c must be a constant



More Big-Oh Examples

- 7n-2

  7n-2 is O(n)

  need $c > 0$ and $n_0 \geq 1$ such that $7n-2 \leq c \cdot n$ 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 $3n^3 + 20n^2 + 5 \leq c \cdot n^3$ for $n \geq n_0$

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

- 3 log n + log log n

  3 log n + log log n is O(log n)

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

  this is true for $c = 4$ and $n_0 = 2$

Big-Oh Rules

- If $f(n)$ is a polynomial of degree d, then $f(n)$ is $O(n^d)$, i.e.,
  1. Drop lower-order terms
  2. Drop constant factors
  - ***$f(n) = 4n^4 + n^3 \Rightarrow O(n^4)$***
- Use the smallest possible class of functions
  - Say "$2n$ is $O(n)$" instead of "$2n$ is $O(n^2)$"
- You can combine growth rates
  - $O(f(n)) + O(g(n)) = O(f(n) + g(n))$
  - $O(n) + O(n^3 + 5) = O(n + n^3 + 5) = O(n^3)$

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

Rules for Analyzing Running Time

- Loops
  - The running time of a loop is at most the running time of the statements inside the loop times the number of iterations

```
for ( x = 0; x < N; x ++ ) {
        statement 1
        statement 2
        …
        statement c
}
```

N iterations
c statements

O(cN) = O(N)

Rules for Analyzing Running Time

- Nested Loops - analyze inside out
  - The running time of a statement inside a group of nested loops is the running time of the statement multiplied by the product of the sizes of all loops

```
for ( x = 0; x < N; x ++ ) {
        for ( y = 0; y < N; y ++ ) {      N*N iterations
                statement 1               1 statements
        }
}                                         O(N*N) = O(N^2)
```


Rules for Analyzing Running Time

- Consecutive statements
  - Just add them together - largest one counts

```
statement 1
statement 2                               2 statements
for ( x = 0; x < N; x ++ ) {
        statement 3                       N iterations
}
                                          O(2+N) = O(N)
```

Rules for Analyzing Running Time

- if/else statements
  - The running time is never more than the running time of the test plus the larger of the running times of S1 and S2

```
if ( condition )          O(running time of condition)
        S1                              +
else                      max ( O(running time of S1),
        S2                      O(running time of S2) )
```


Analyzing Running Time

- In class exercise - give the Big-Oh running time of the following code

```
for ( x = 0; x < N; x ++ )
        array[x] = x*N;

for (x = 0; x < N; x ++ ) {
        if ( x < (N/2) )
                cout << array[x];
        else
                for (  y = 0; y < N; y ++ )
                        cout << y*array[x];
}
```

Calculating Big-Oh

- In class exercise - Give the Big-Oh notation for the following functions:
  - $n + \log(n) =$
  - $8n \log(n) + n^2 =$
  - $6n^2 + 2^n + 300 =$
  - $n + n \log(n) + \log(n) =$
  - $40 + 8n + n^7 =$

Real World Impact

Suppose each operation takes 1 nanoseconds (10^{-9} seconds)

n	lg n	n	n lg n	n^2	2^n	n!
10	0.003μs	0.01μs	0.033μs	0.1μs	1μs	3.63ms
20	0.004μs	0.02μs	0.086μs	0.4μs	1ms	77.1years
30	0.005μs	0.02μs	0.147μs	0.9μs	1sec	$>10^{15}$years
100	0.007μs	0.1μs	0.644μs	10μs	$>10^{13}$years	
10,000	0.013μs	10μs	130μs	100ms		
1,000,000	0.020μs	1ms	19.92 ms	16.7min		

- For $n < 10$, the difference is insignificant.
- $\Theta(n!)$ algorithms are useless well before $n = 20$.
- $\Theta(2^n)$ algorithms are practical for $n < 40$.
- $\Theta(n^2)$ and $\Theta(n \lg n)$ are both useful, but $\Theta(n \lg n)$ is significantly faster.

Array Big Oh Running Times

- Unsorted insert
  - O(1) - add to end
- Sorted insert
  - O(N) - shift items
- Num items
  - O(1) - have to keep counter
- Print
  - O(N)
- Sorted Remove
  - O(N) - shift items
- Unsorted Remove
  - O(1) - move last
- Linear search
  - O(N)

Linked List Running Times

- Assuming only a head pointer and a singly linked list
  - Insert head
  - Insert tail
  - Search
- Remove
- Num items
- Print

Space Complexity

- Similar to determining Big-Oh runtime complexity
- Give upper bound on space required based on the input size
  - Constant factors and low-order terms are not significant

Math you need to Review

- Logarithms and Exponents
  - **properties of logarithms:**

    $\log_b(xy) = \log_b x + \log_b y$
    $\log_b (x/y) = \log_b x - \log_b y$
    $\log_b x^a = a\log_b x$
    $\mathbf{\log_b a = \log_x a / \log_x b}$
  - **properties of exponentials**:

    $a^{(b+c)} = a^b a^c$
    $a^{bc} = (a^b)^c$
    $a^b / a^c = a^{(b-c)}$
    $b = a^{\log_a b}$
    $b^c = a^{c*\log_a b}$