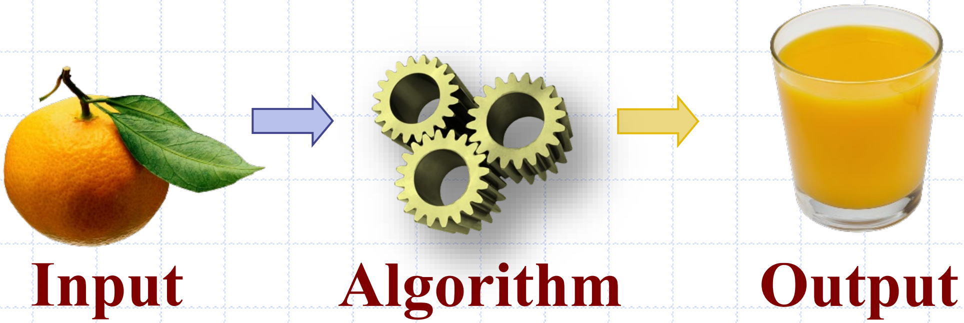
O(f(n))
- order- how complex

Analysis of Algorithms



Sareh Taebi

FLORIDA ATLANTIC UNIVERSITY

Which solution is more efficient in terms of number of operations?

```
## R-1.2 ##
def is_even(k):
 i = k
 while i != 0 and i != 1:
  i = i - 2 if i > 0 else i + 2
 if i == 0:
  print(k, "is even.")
  return True
 if i == 1:
  print(k, "is odd.")
  return False
```

```
def isevenpt1(n):
 z = str(n)
 x = int(z[-1])
 return isevenpt2(x)
def isevenpt2(n):
 x = 0
 while x < n:
 x += 2
 return x == n
```

```
def is_even(k):
return k & 1 == 0
```

testing efficiency of each program
- time each one

(Taken from assignment 1 submissions)

Finding largest and smallest Values in a set

```
# Function for R-1.3
def minmax(data):
        mini = maxi = data[0]
        for i in range(len(data)):
        if data[i] < mini:
                mini = data[i]
        if data[i] > maxi:
                maxi = data[i]

        return (mini,maxi)
```

guess 2nd is more efficient

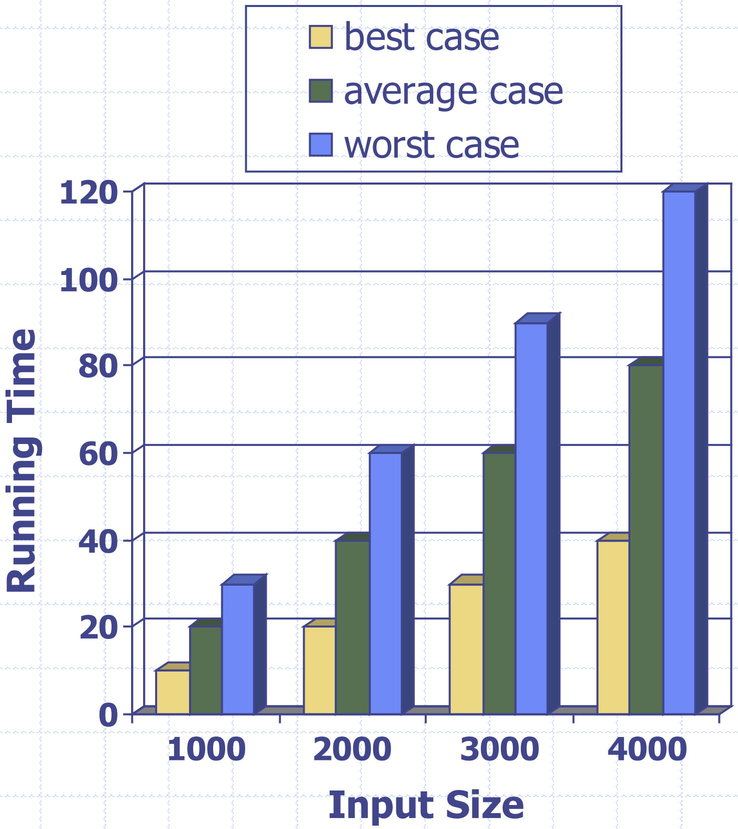
```
def minmax(data):
        largest = data[0]
        smallest = data[0]
        for n in data:
                if n > largest:
                        largest = n
                elif n < smallest:
                        smallest = n
        return smallest, largest
```

(Taken from assignment 1 submissions)

Running Time

how fast algorithm will be as input increases
prepare for the worst case

- 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

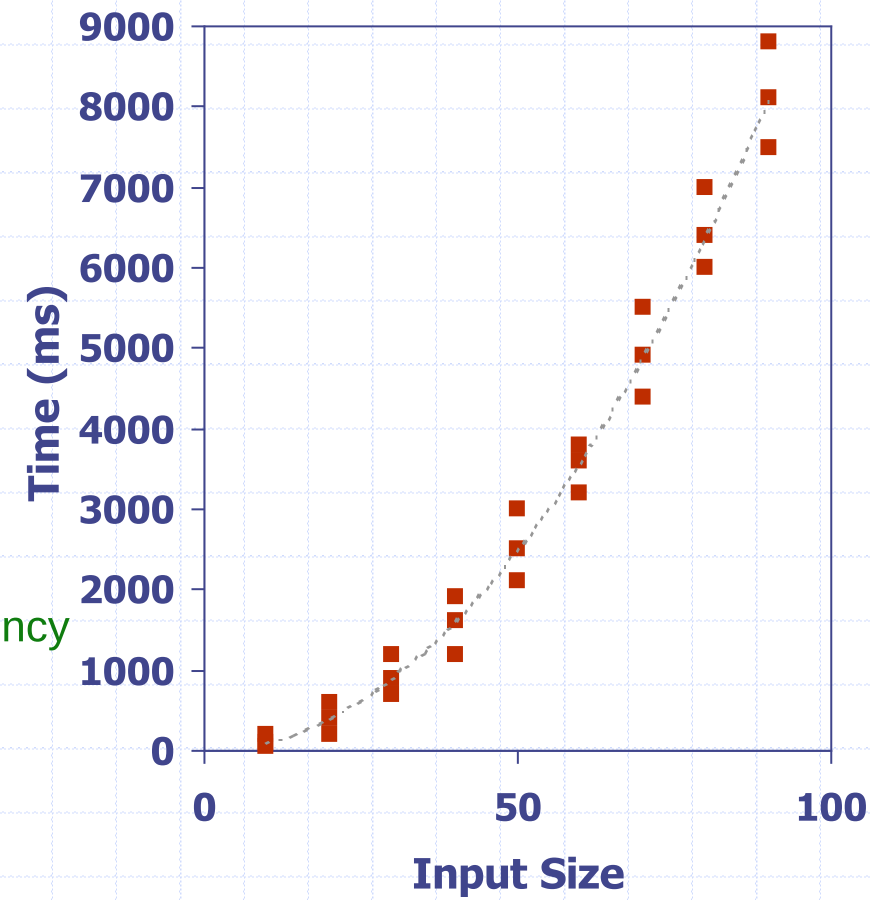


Experimental Studies

- 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
```

compare efficiency



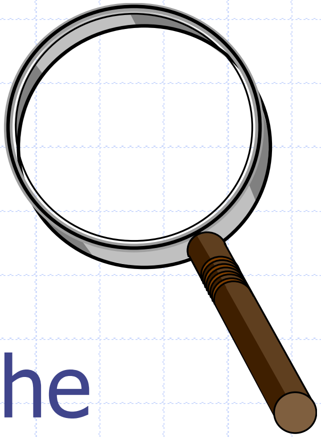
- Plot the results

Limitations of Experiments

algorithm without implementation

- 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*.

f(n) function that characterizes running time

- Takes into account all possible inputs
- Allows us to evaluate the speed of an algorithm independent of the hardware/software environment

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

Pseudocode for finding max value in a list of integers

every line is a task

function in the order of n
linear increase better than
exponential increase

costly operations:
- comparisons
- multiplication
- division

not costly:
- variable assignments

```
max (list of a integers):
    max = a1 0
    for i = 1 to n-1  (i<n)
     if max < ai then max = ai
     return max
```

How many costly operations are done?
n - 1: the max < ai comparison is made n -1 times.
n - 1 : each time i is incremented a test is done to see if i <n
1: one last operation determines that i >= n

2n – 1 operations

Seven Important Functions

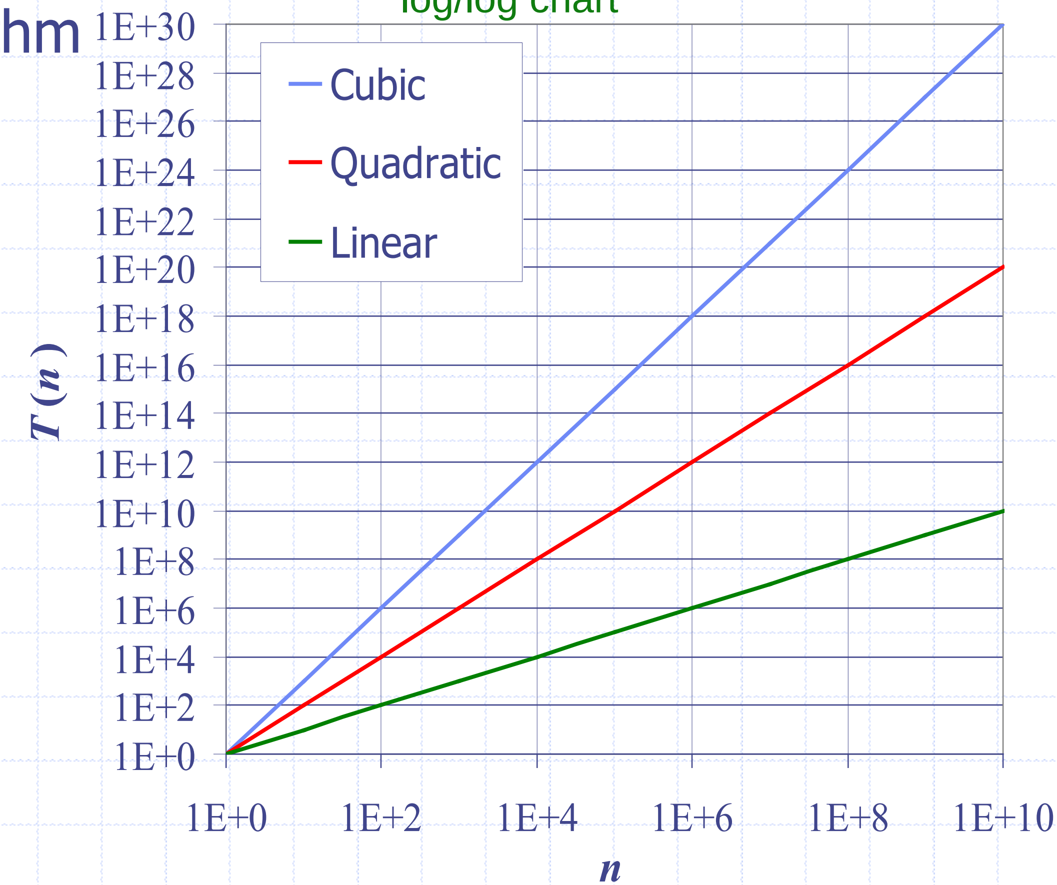
- Seven functions that often appear in algorithm analysis:

f(n) = constant value

  - 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

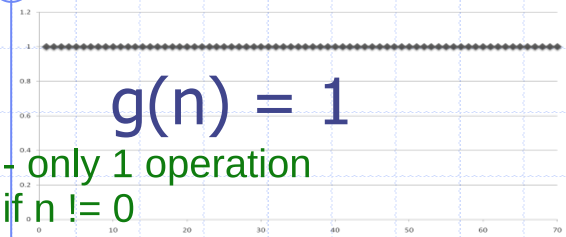
download module for plotting log/log chart



slope = rate of growth of algorithm

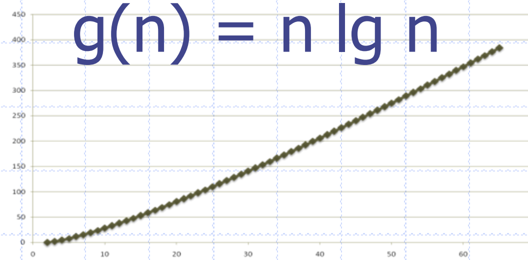
Functions Graphed Using “Normal” Scale

Slide by Matt Stallmann included with permission.

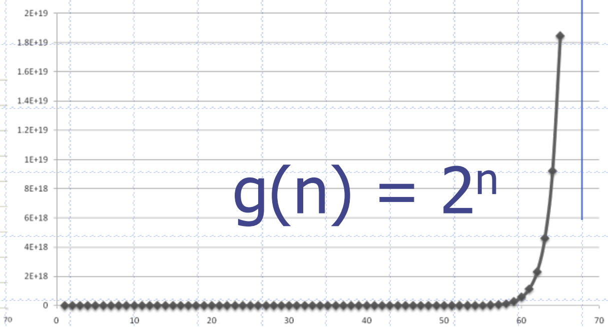


$g(n) = 1$

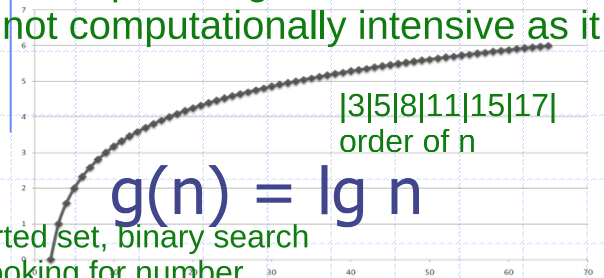
- only 1 operation
if n != 0
  return true
or simple assignment
not computationally intensive as it grows



$g(n) = n \lg n$



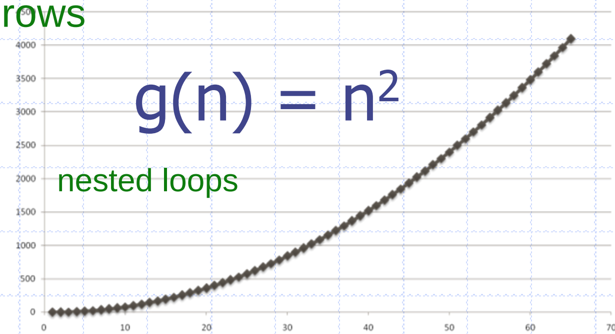
$g(n) = 2^n$



$g(n) = \lg n$

|3|5|8|11|15|17|
order of n

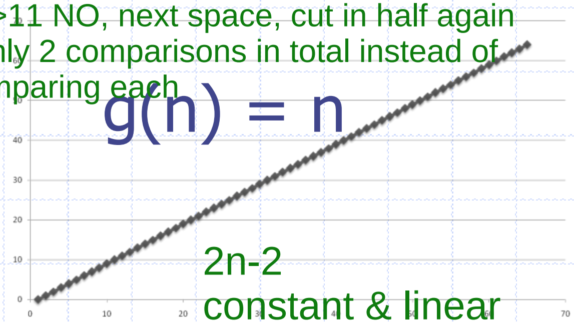
sorted set, binary search
- looking for number
- divide space in half
- compare 11 to place of division
- 8>11 NO, next space, cut in half again
- only 2 comparisons in total instead of comparing each



$g(n) = n^2$

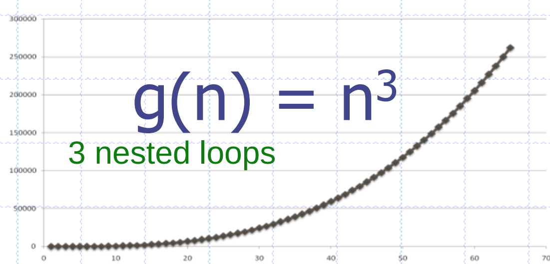
nested loops

functions
- evaluation of complexity for algorithm



$g(n) = n$

2n-2
constant & linear
complexity in order of n



$g(n) = n^3$

3 nested loops

Primitive Operations

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

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

actual code in book

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

```
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
```

- Step 1: 2 ops, 3: 2 ops, 4: 2n ops, 5: 2n ops, 6: 0 to n ops, 7: 1 op