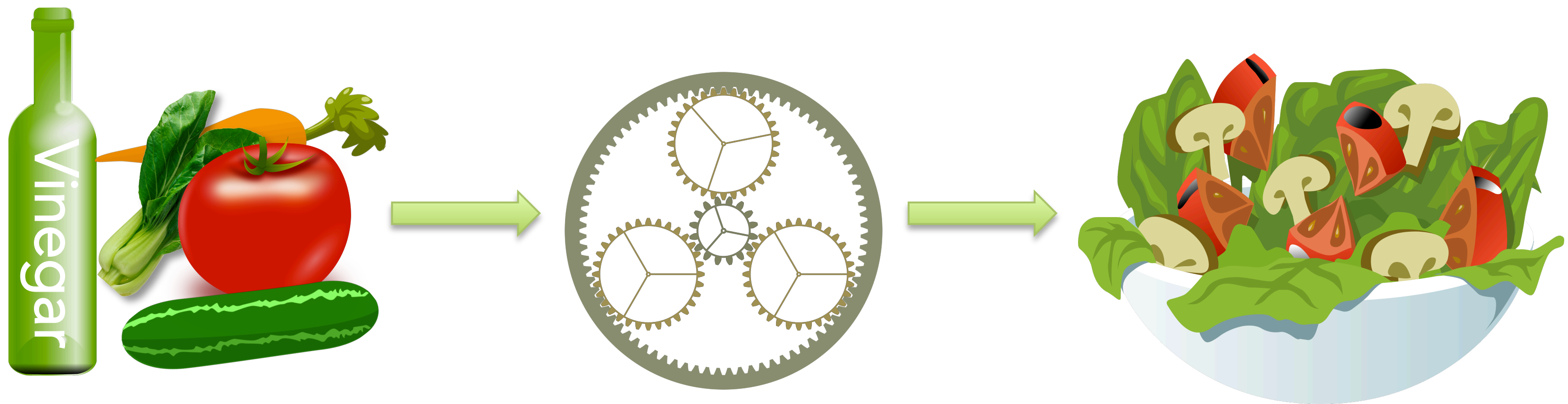
Lecture 5
Algorithm Analysis

Which data structure is good for the given problem?

Algorithm



Each data structure enables certain algorithms!

How to characterize an algorithm?

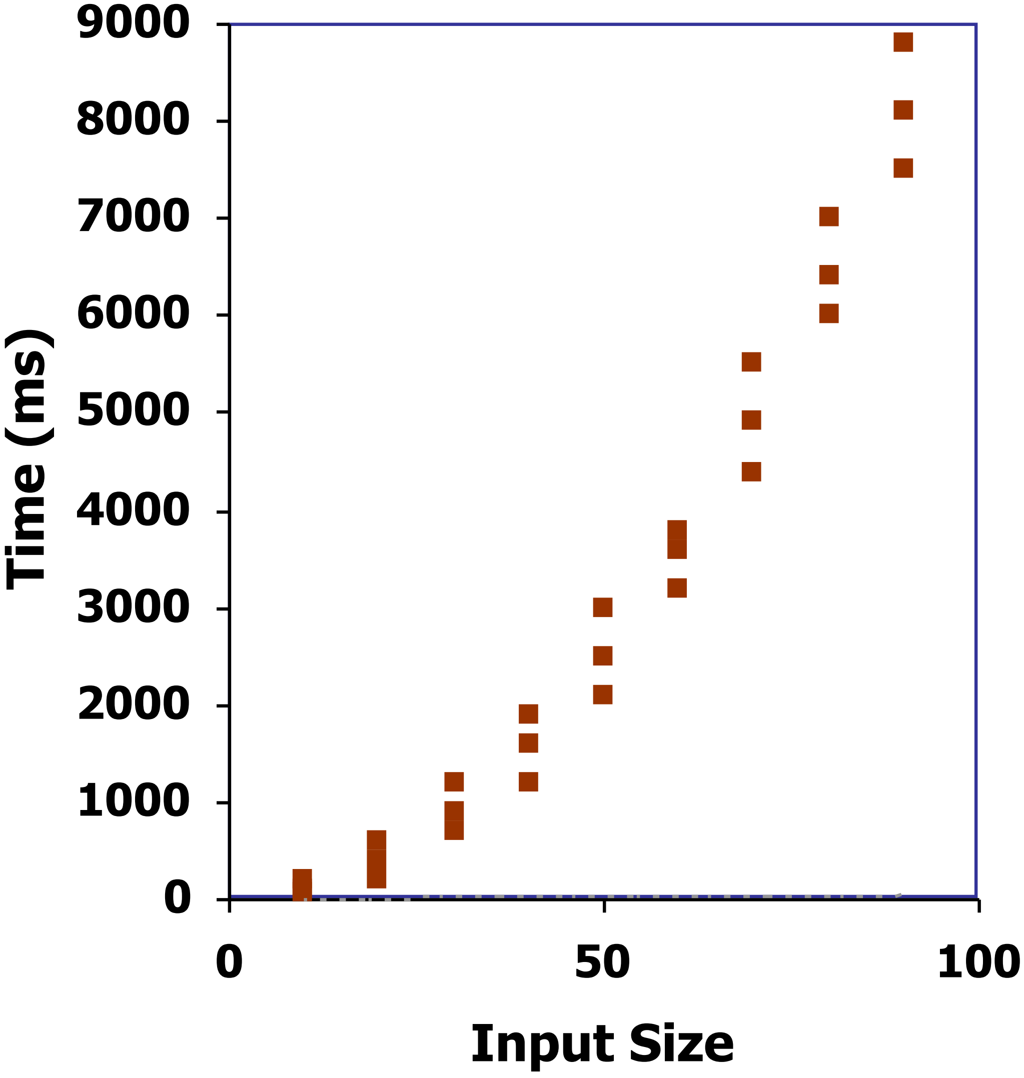
Some criteria

1. Correctness
2. Time efficient
3. Memory efficient
4. Coding efficient

How to measure the time efficiency of an algorithm?

Empirical: Run and Record

- Implement the algorithm
- vary the input size
- use function like clock() to record
- plot input size vs time



9000
8000
7000
6000
5000
4000
3000
2000
1000
0
Time (ms)
0
50
100
Input Size

Limitations

- need to implement
- need time to run for all inputs
- same hardware and software environment required

Theoretical Analysis

- only needs high level description
- platform independent
- no implementation required

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

```
if condition then
    true-actions
else
    false-actions
```

```
while condition do
    actions
```

```
repeat
    action
until condition
```

Primitive operations

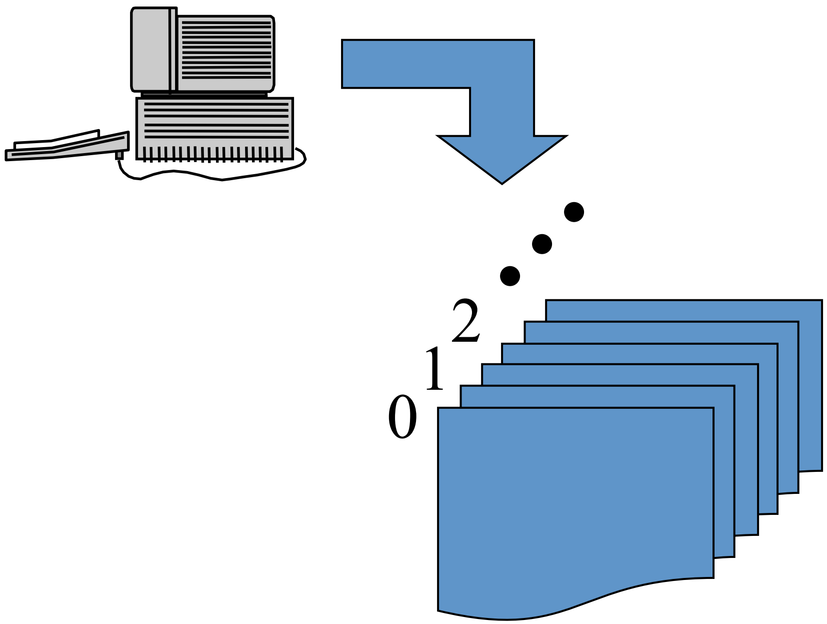
- Basic computations, e.g. assignment, evaluating expression, array indexing
- call function /return from a function

Main Assumptions

- sequential execution
- primitive operations take fairly similar time to execute
- [RAM MODEL] accessing elements takes constant time

Random Access Machine

- A CPU
- Unbounded memory
- Unit access time



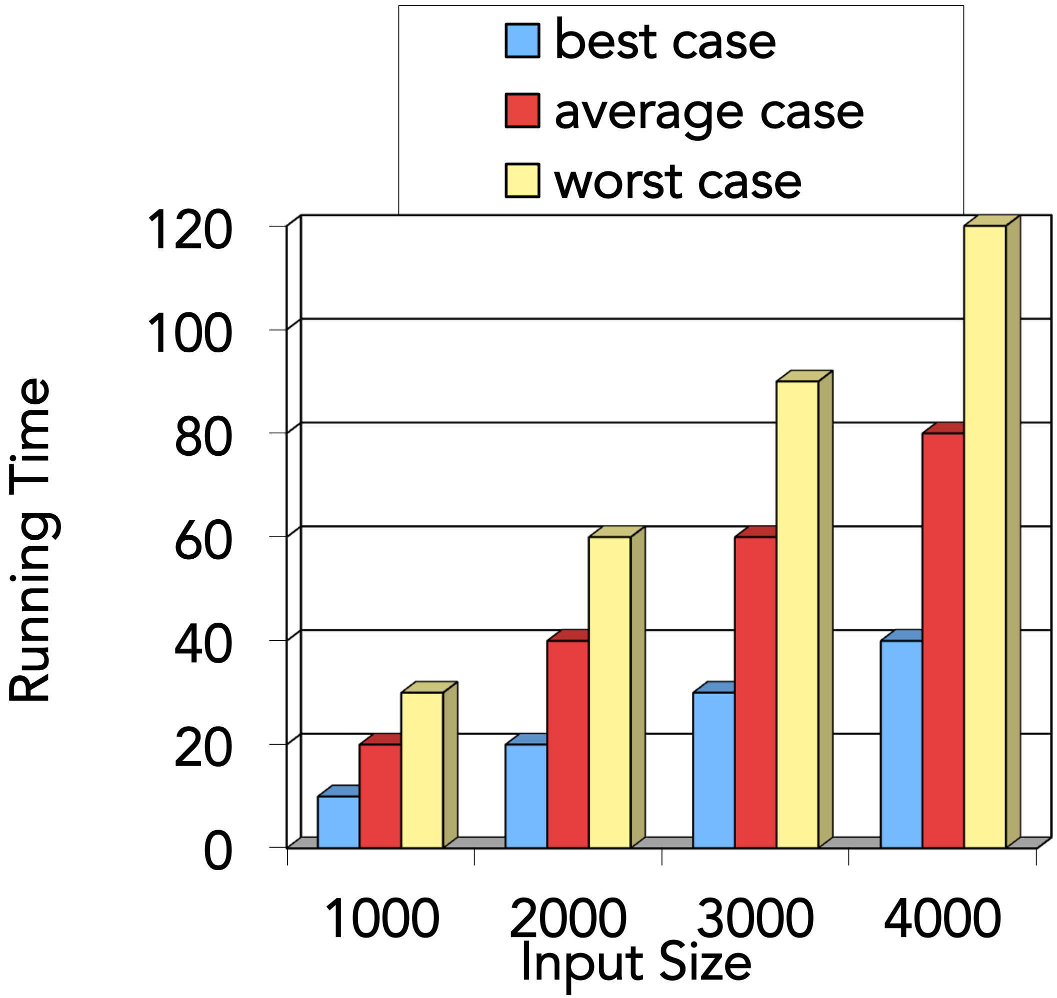
counting number of primitive operations

Algorithm *arrayMax*(A, n)	# operations
$currentMax \leftarrow A[0]$	2
for $i \leftarrow 1$ to $n - 1$ do	$2n+2$
if $A[i] > currentMax$ then	$2(n - 1)$
$currentMax \leftarrow A[i]$	0 to $2(n - 1)$
return *currentMax*	1

worst case = $6n+1$ best case = $4n+3$

Worst case	Best case
$T\ (n) = 6n+1$	$T(n) = 4n+3$

What is the average case?



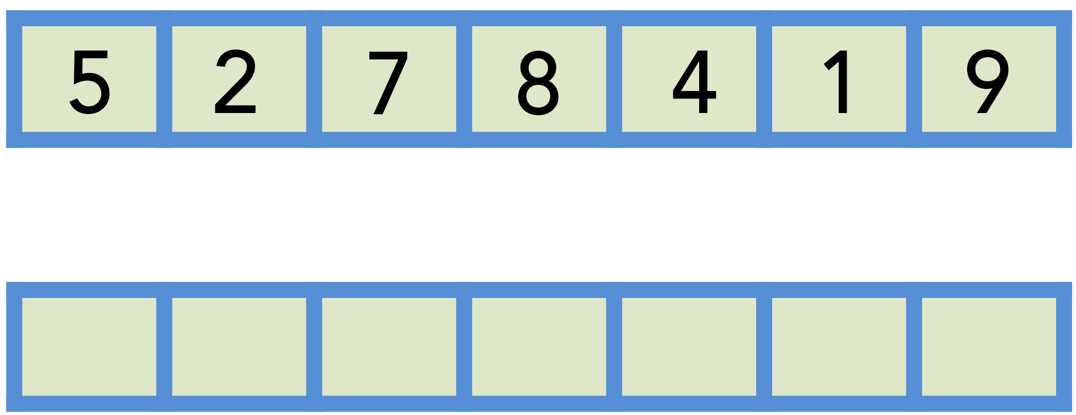
best case
average case
worst case
120
100
80
60
40
20
0
Running Time
1000
2000
3000
4000
Input Size

We will characterize running time in terms of worst case!

What is the worst case running time of insertion sort?

Insertion Sort

Insert elements at right place one by one!



Insertion Sort

```
for j=2 to n
  key = A[j];
  //Insert A[j] into sorted sequence A[1…j-1]
  i=j-1
  while i > 0 and A[i] > key
    A[i+1]=A[i]
    i=i-1;
A[i+1] = key;
```

Which one takes longer?

```
void pushAll (int k){
  for (int i=0; i<= 100*k; i++)
   {
    list.add(i);
    }
 }
```

```
void pushAdd(int k) {
  for (int i=0; i<= k; i++){
    for (int j=0; j<= k; j++){
      list.add(i+j);
    }
  }
}
```

100K add operations

K^2 add operations

Which grows faster?

$f(k)=100K$	$f(k)=k^2$
$f(0) = 0$	$f(0)=0$
$f(1)=100$	$f(1) = 1$
$f(100) = 10^4$	$f(100)=10^4$
$f(1000)=10^5$	$f(1000)=10^6$

Growth is more important than actual running time!

Growth Rate of Running Time

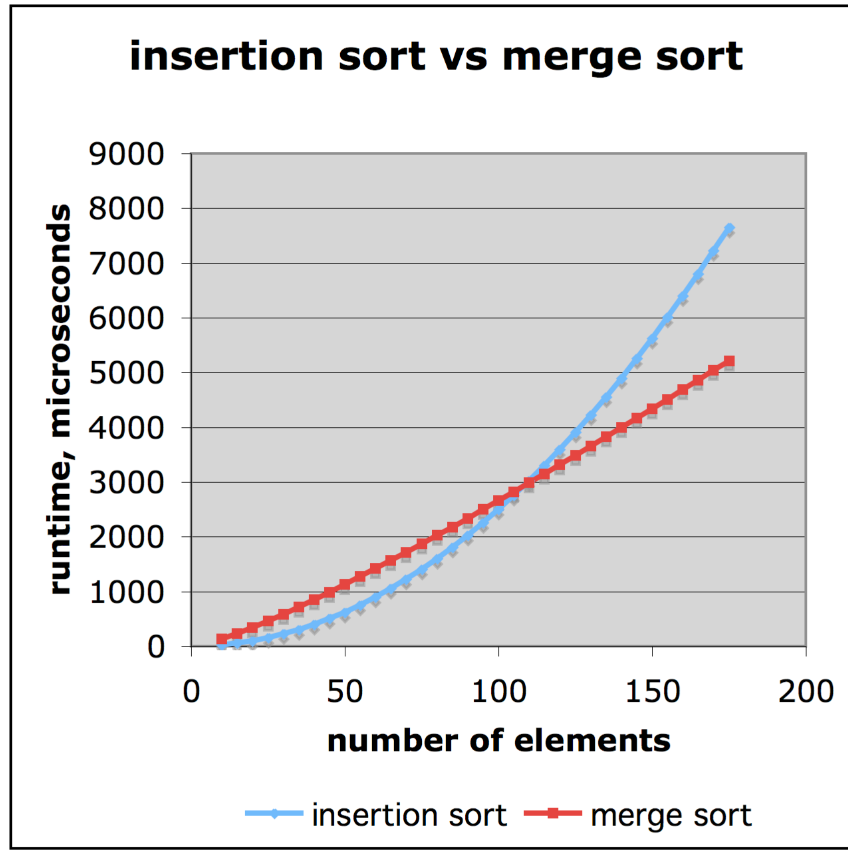
- hardware/ software environment
  - affects by a constant factor
  - does not alter the growth rate
- growth rate is an intrinsic property of algorithm

Comparing Two Algorithms

To sort 1 million items:

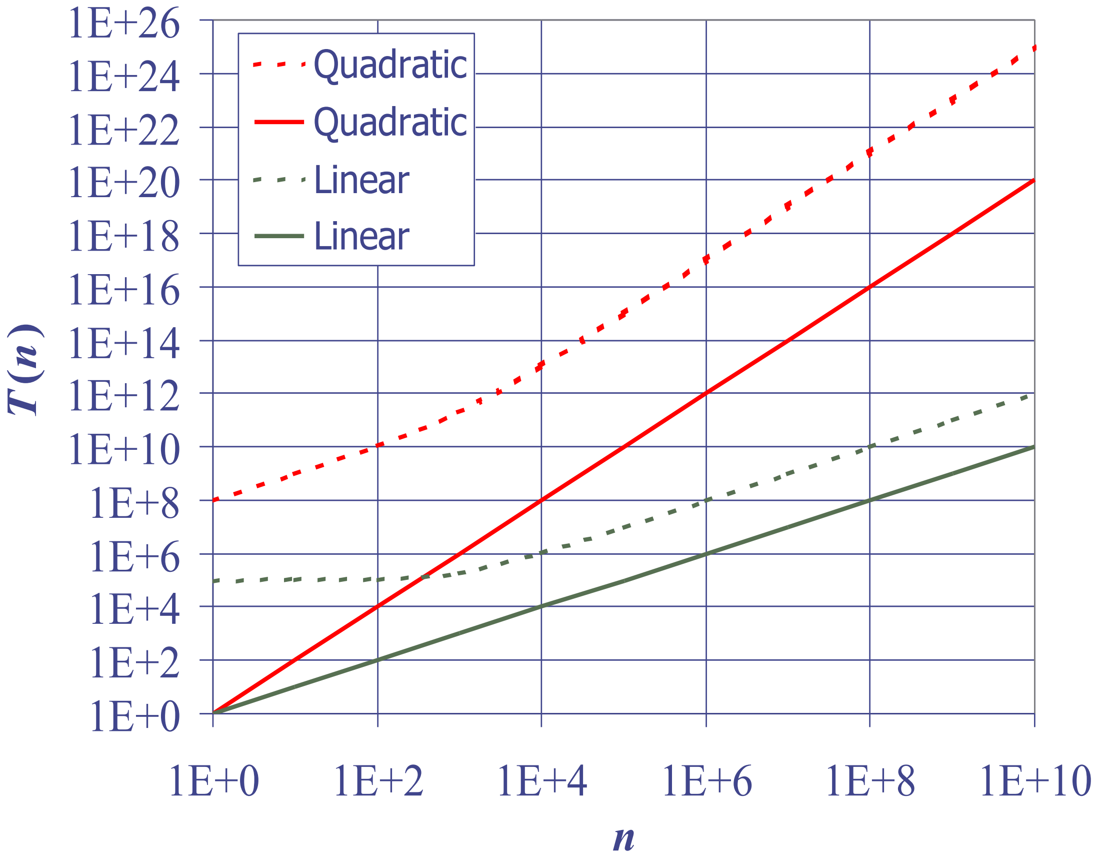
-insertion sort ($n^2/4$) takes 70 hours

-merge sort(n log n) takes 40 seconds



Growth rate is not affected by constant or lower order terms!

- 10^2n+10^5
- $105n^2 + 10^8n$

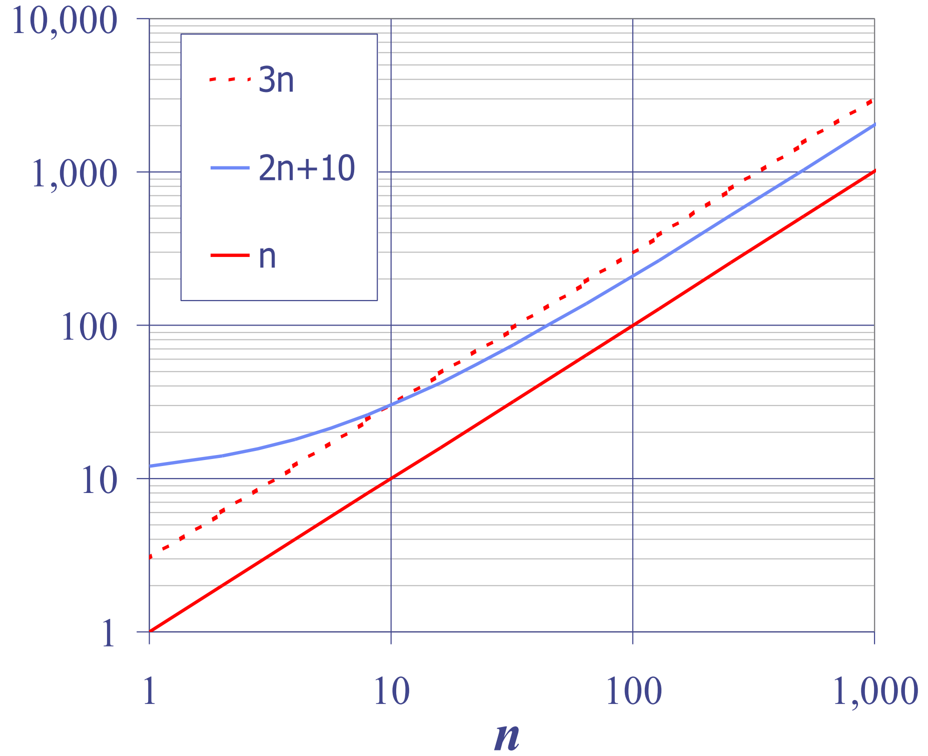


Asymptotic Notation

Big-Oh

We say $f(n)$ is $O(g(n))$ if $f(n) \leq cg(n)$ for some c and $n > n_0$!

E.g. 2n+10 is O(n), how?



Big-Oh notation allows us to ignore constant factors and lower order terms!

Look for simplest terms for expressing Big-Oh!

Big-Oh Examples

- $7n-2$ 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 $n_0 = 1$

- $3n^3 + 20n^2 + 5$ 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 $n_0 = 21$

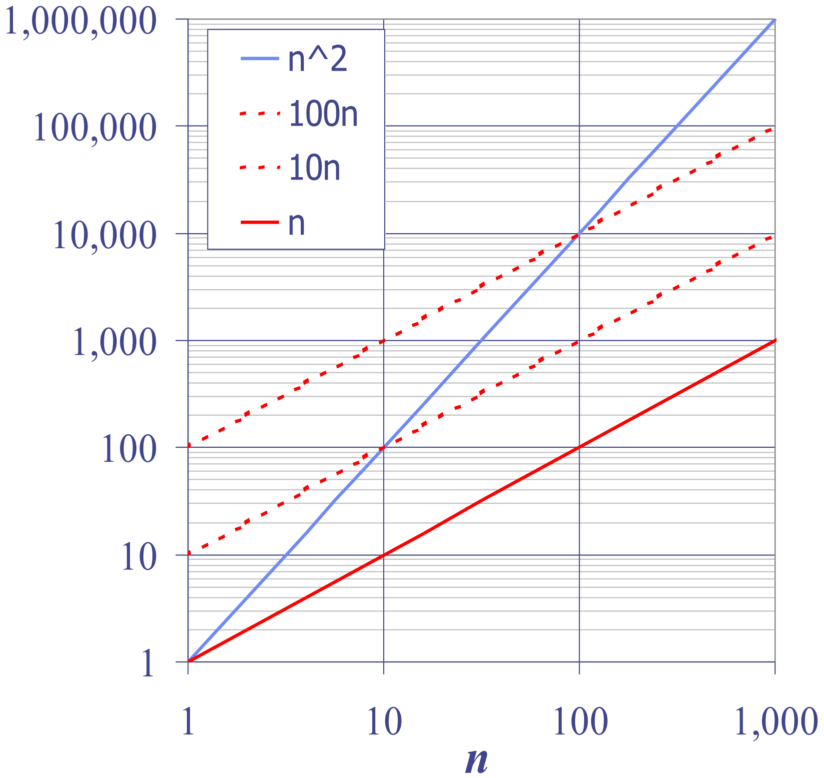
- $3 \log n + 5$ is $O(\log n)$

  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 $n_0 = 2$

Not Big-Oh Example

- the function n^2 is not $O(n)$
- $n^2 \le cn$
- not possible



Big-Oh and Growth Rate

- gives an upper bound
- $f(n)$ is $O(g(n))$ tells f(n) does not grow faster than $g(n)$
- Can be used to compare algorithms

Focus on the main factor that determines the growth rate!

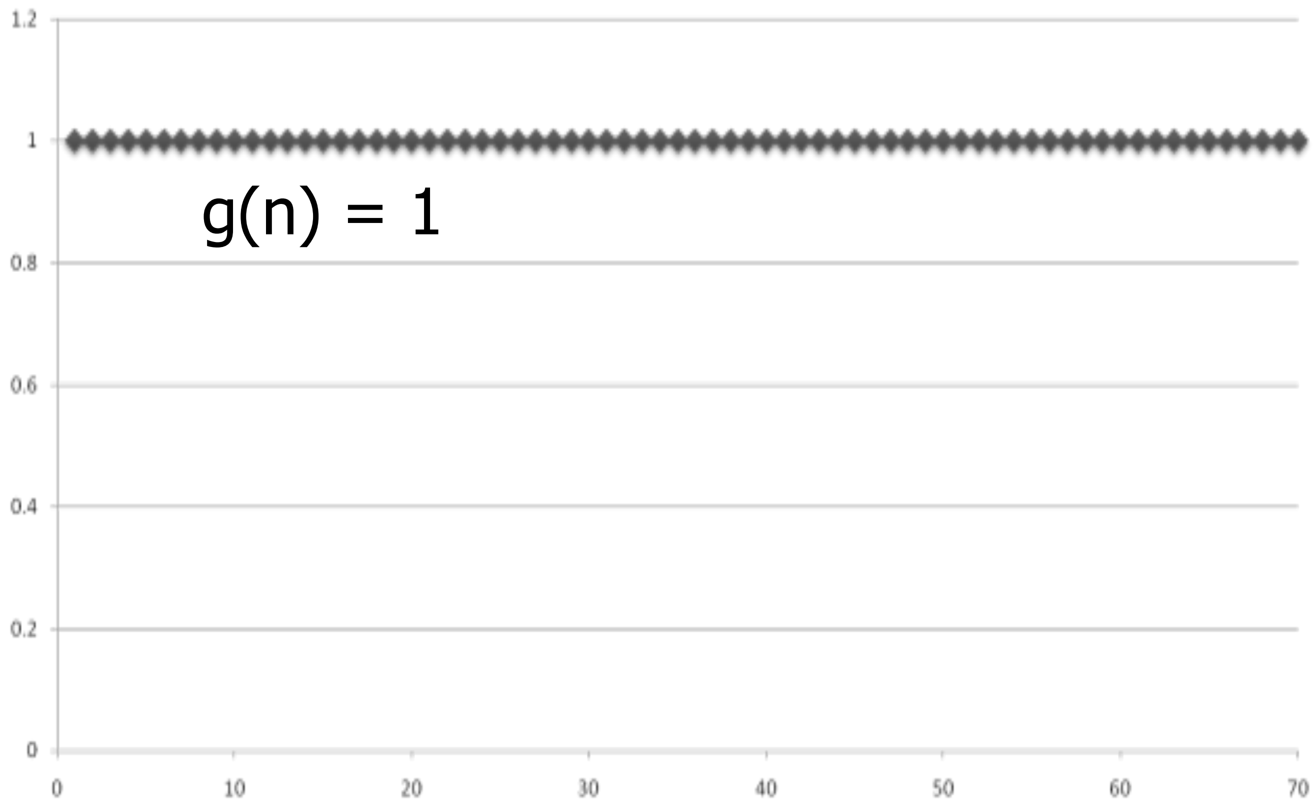
Running time grows proportional to a "specific" function of n within a constant factor!

The Seven Important Functions

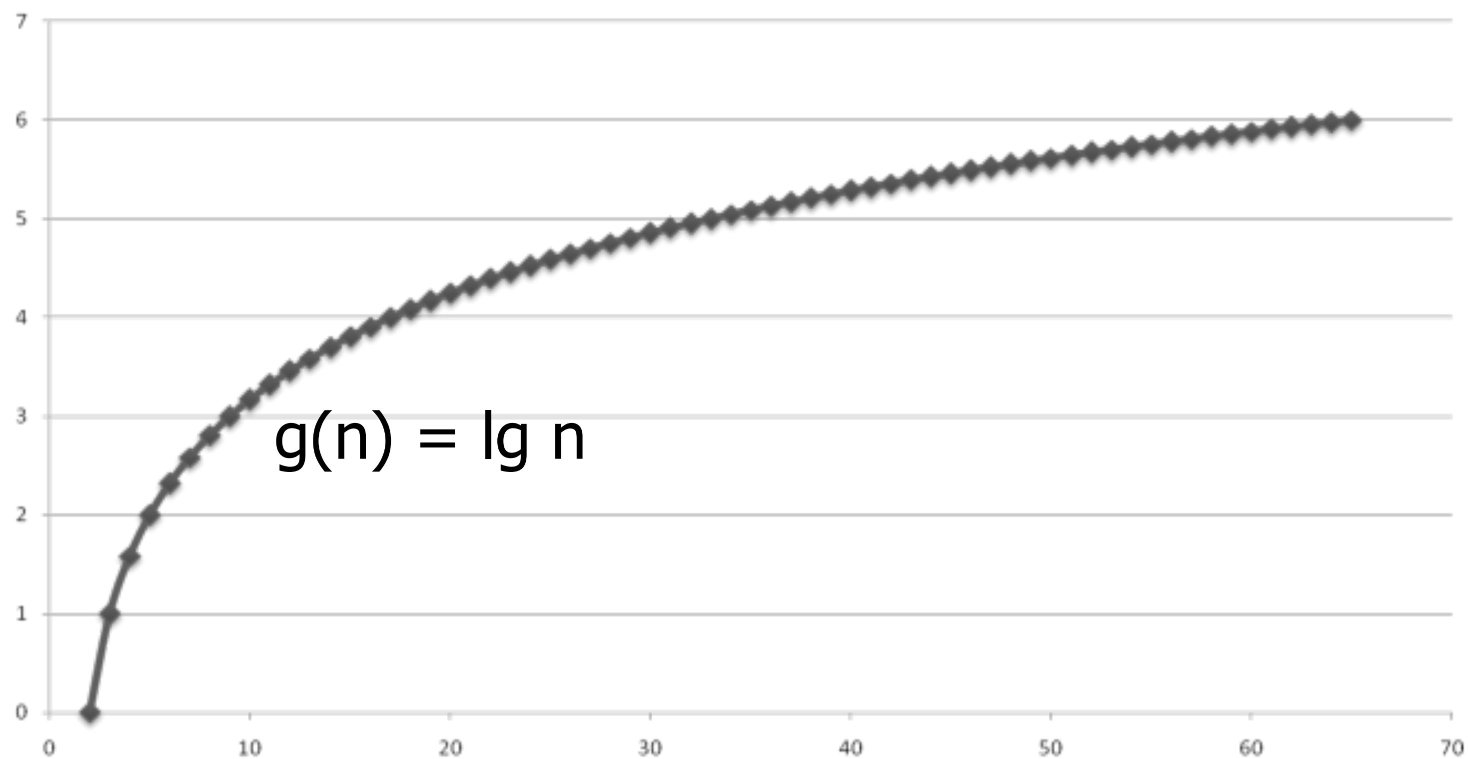
Some Important Functions

- Constant ≈ 1
- Logarithmic ≈ log n
- Linear ≈ n
- N-Log-N ≈ n log n
- Quadratic ≈ n^2
- Cubic ≈ n^3
- Exponential ≈ 2^n

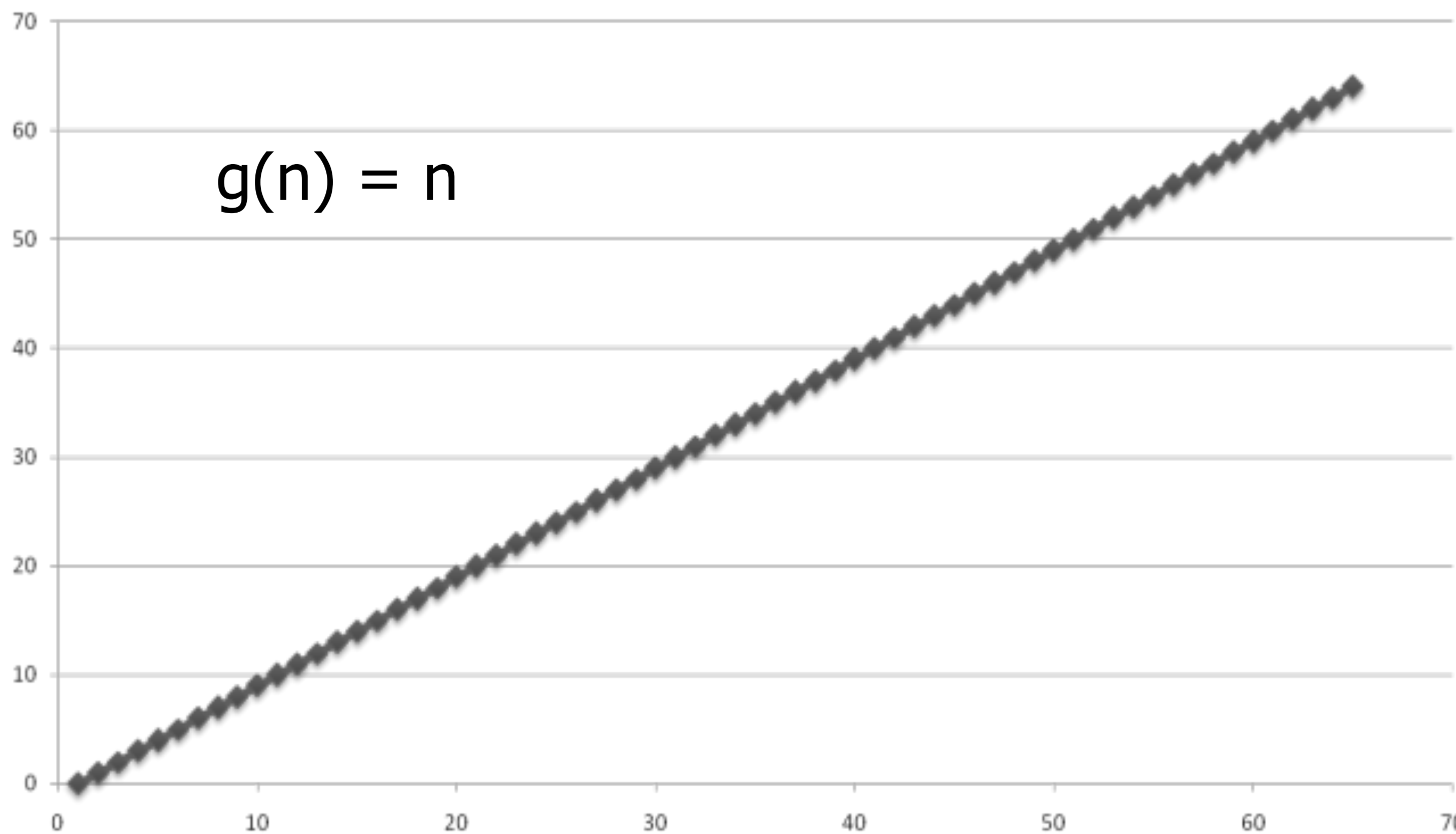
Constant ≈ 1



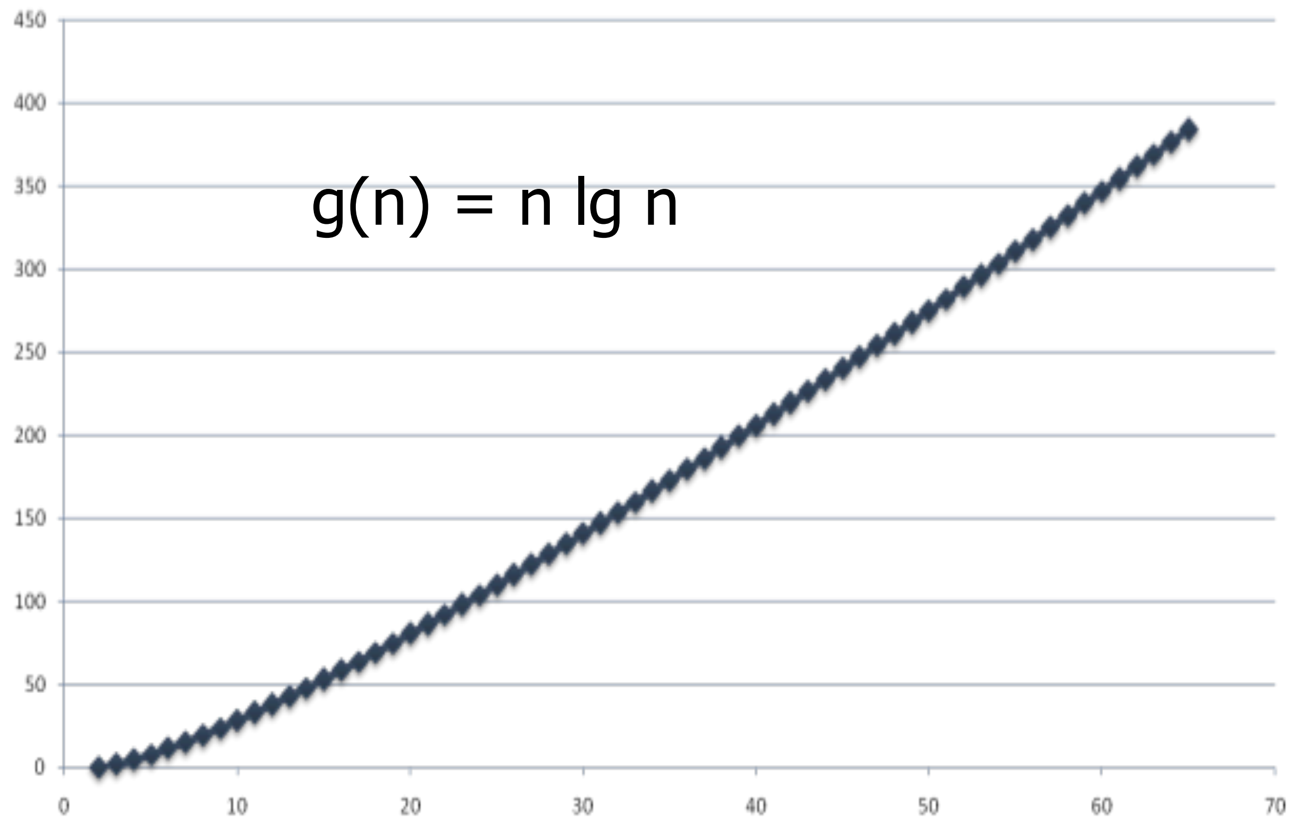
Logarithmic ≈ log n



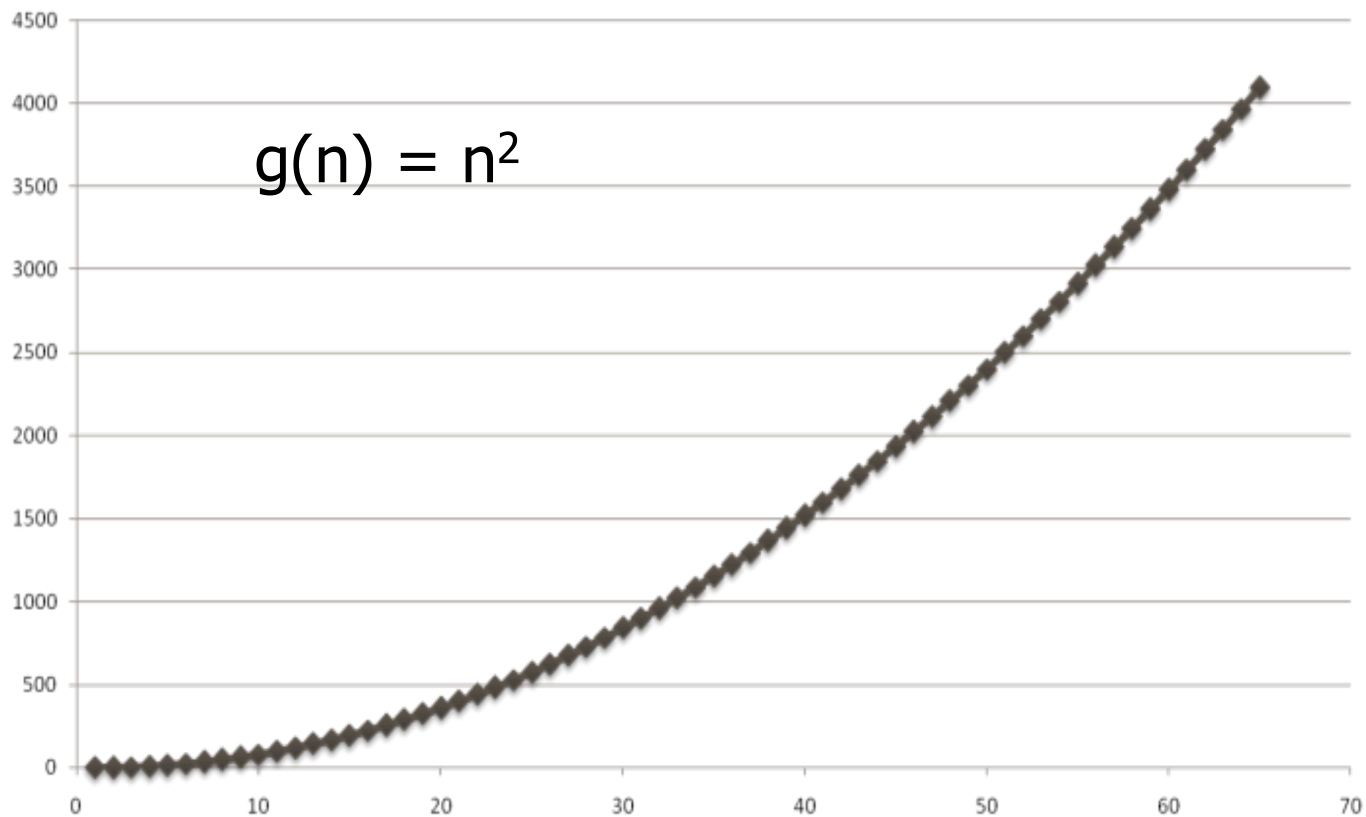
Linear ≈ n



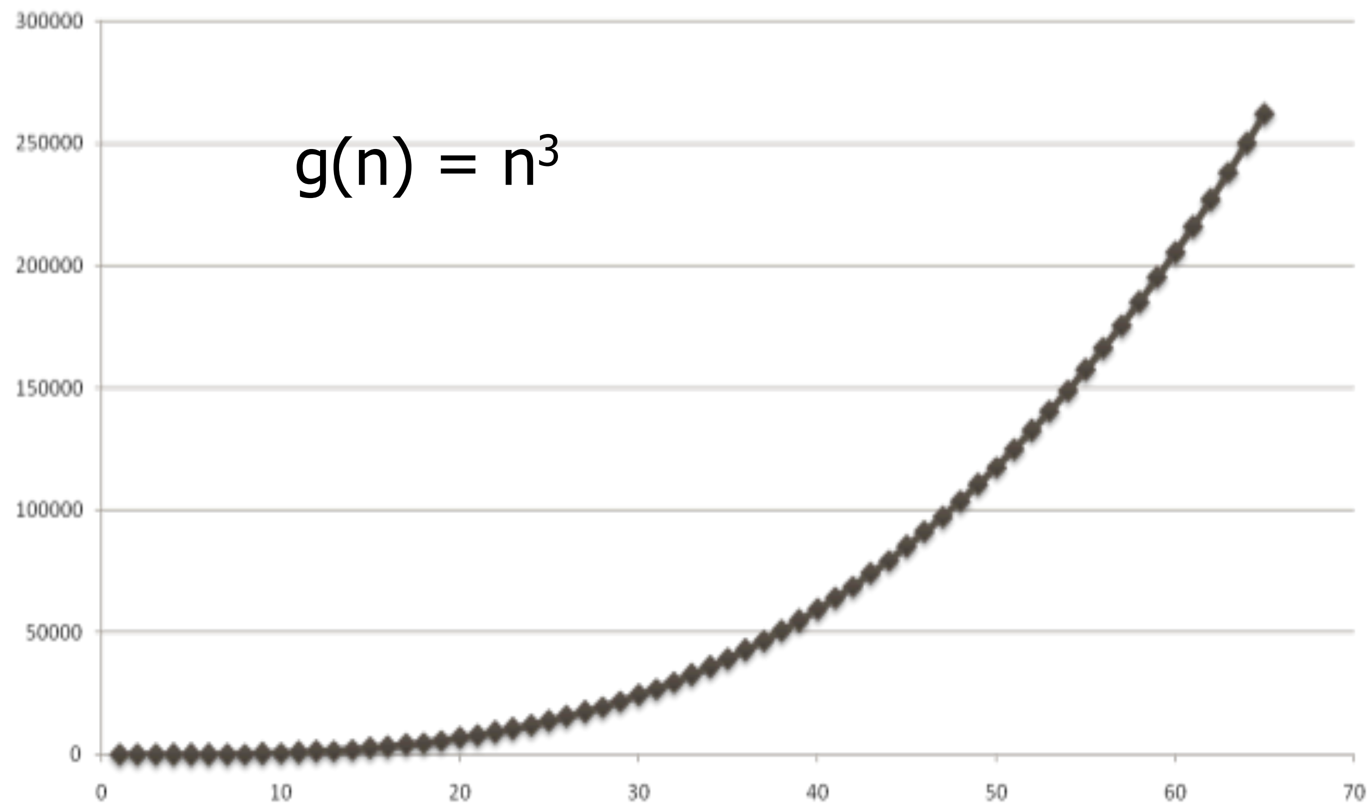
N-Log-N ≈ n log n



Quadratic ≈ n^2

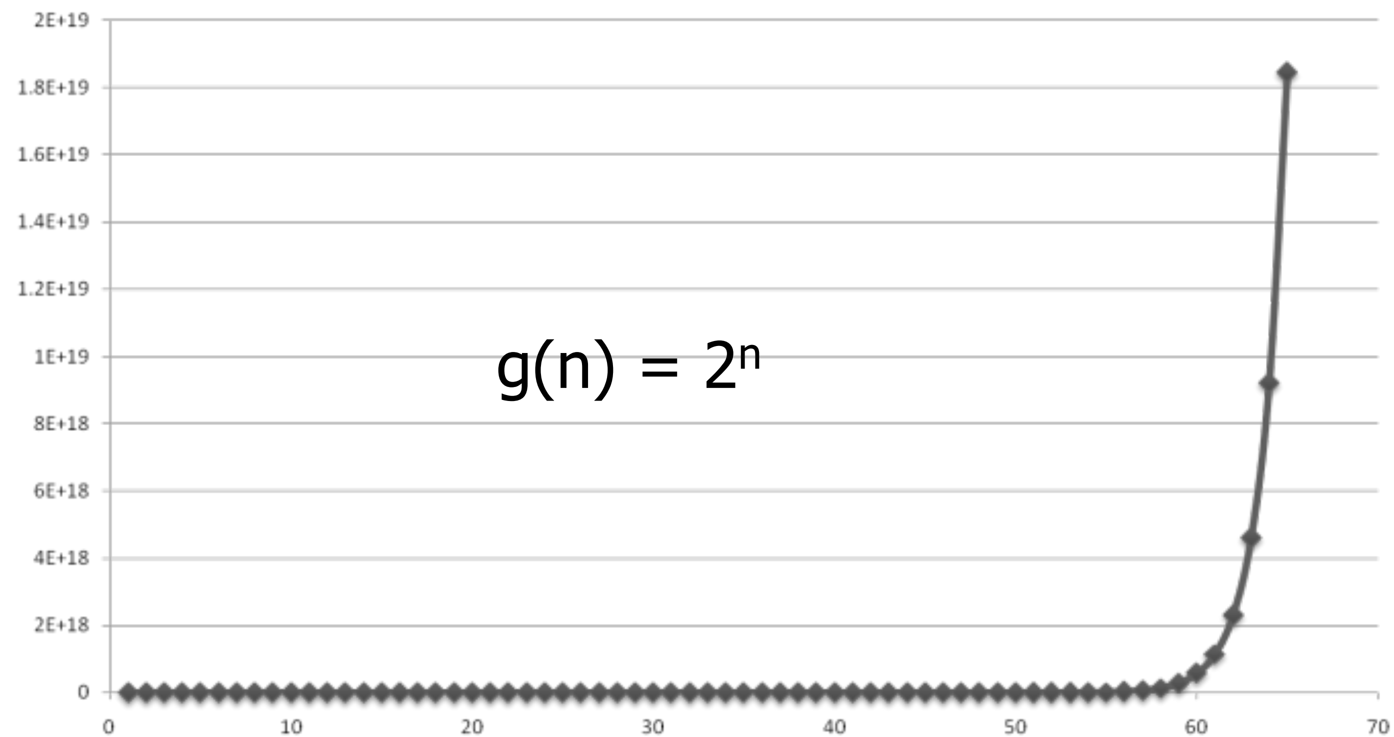


Cubic ≈ n^3

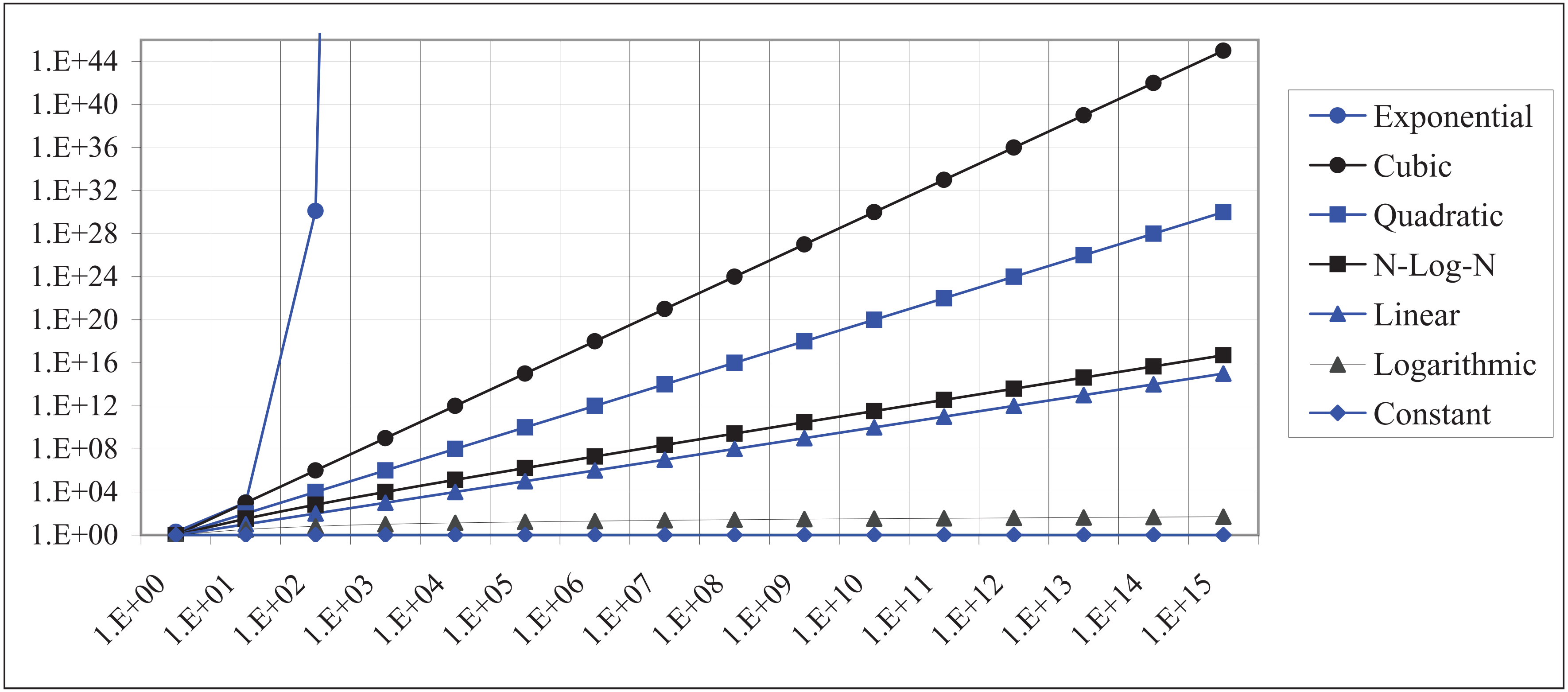


g(n) = n3
300000
250000
200000
150000
100000
50000
0
0
10
20
30
40
50
60
70

Exponential ≈ 2^n



Growth rate on Log Scale



Asymptotic Analysis

1. measure running time in terms of input
2. calculate Big-Oh of the function

Example

- worst case running time of arrayMax
  - $f(n) = 6n+1$
- Big-Oh of f(n) if n, i.e.
  - $6n+1$ is $O(n)$
- constant factors can be ignored while counting primitives itself

Rules of thumb

Polynomial Runtime

- Drop lower order terms
- Drop constants
- $f(n) = a_k n^k + a_{k-1} n^{k-1} \ldots + a_0$
  - $f(n)$ is $O(n^k)$

Loops

cost = (#iterations)×(#max cost of one iteration)

```
int sum (int A[], int n){
  int total=0;
  for (int i=0; i<= n; i++)
   {
    total=total+A[i];
    }
  return total;
 }
```

$O(n)$

Nested Loops

cost = (#iterations)×(#max cost of one iteration)

n iterations

```
int sum (int A[][], int n){
  int total=0;
  for (int i=0; i<= n; i++)
   {
      for (int j=0; i<= n; j++)
          total=total+A[i][j];
   }
  return total;
}
```

$O(n^2)$

Sequential Statements

cost = (#cost of first)+(#cost of second)

```
int sum (int A[], int B[], int n){
  int totalA=0; int totalB=0;
  for (int i=0; i<= n; i++){
      totalA=totalA+A[i];
   }
  for (int j=0; i<= n; j++){
          totalB=totalB+B[j];
  }
  return totalA+totalB;
}
```

cost of most costly step matters, i.e., O(n)

if/else

cost = max(cost of first, cost of second)

```
void sum (int A[], int n){
  int total=0;
  for (int i=0; i<= n; i++)
   {
       if (i%2==0)
            //first action
       else
            // second action
     }
 }
```

O(n*max)

Line between efficient and inefficient?