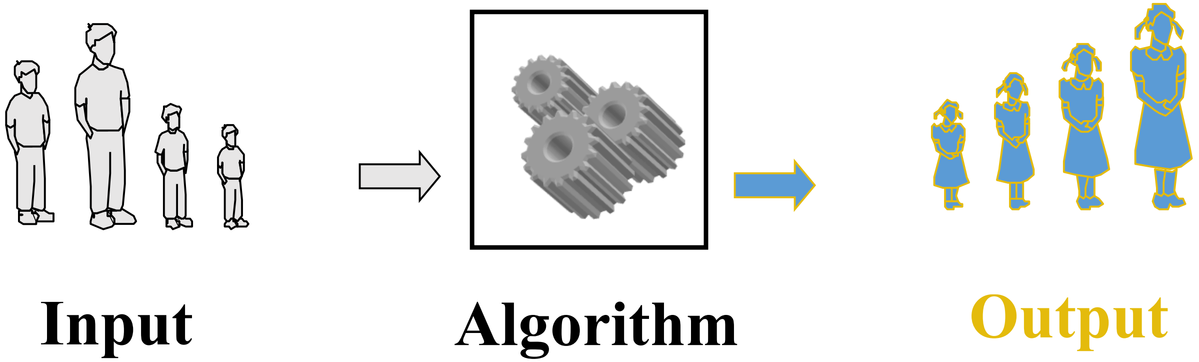
Algorithm



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

Forms of Algorithms

Algorithm Descriptions

- Nature languages: Chinese, English, etc.
- Pseudo-code: codes very close to computer languages, e.g., C programming language.
- Programs: C programs, C++ programs, Java programs.

Why algorithm?

Goal:

- Allow a well-trained programmer to be able to implement.
- Allow an expert to be able to analyze the running time.

What do you expect?

- You will be able to evaluate the quality of a program
- You will be able to write fast programs
- You will be able to solve new problems
- You will be able to give non-trivial methods to solve problems.

Efficiency of Algorithms

Question: How can we characterize the performance of an *algorithm* ...

- Without regard to a *specific computer?*
- Without regard to a *specific language?*
- Over a wide *range of inputs?*

Desire: Function that describes *execution time* in terms of *input size*

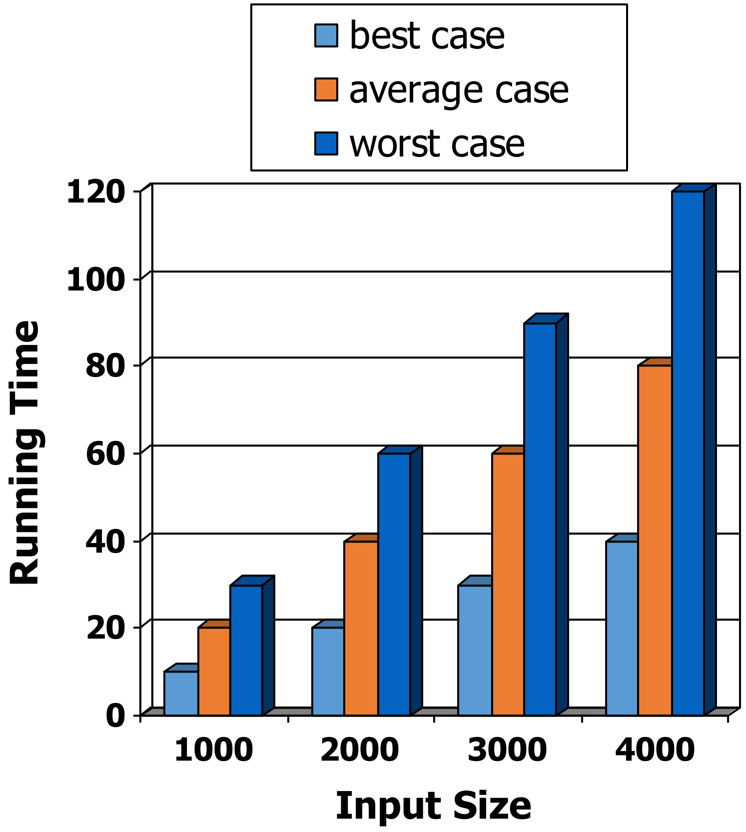
- Other measures might be memory needed, etc.

Performance measurement?

- Estimate the running (execution) time
- Estimate the memory space required.
- Depends on the input size

Running time of an algorithm

- Most algorithms transform input objects into output objects.
- The running time of an algorithm typically grows with the input size.



Counting Primitive Operations

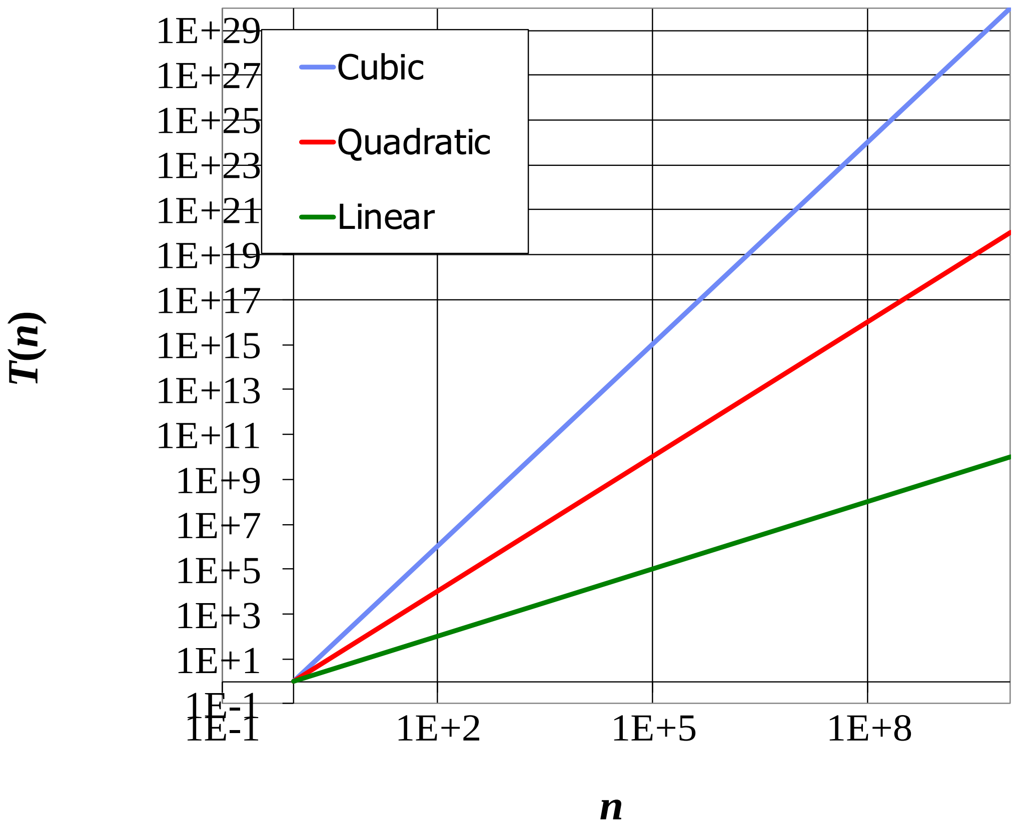
- By inspecting the *pseudo code*, we can determine the maximum number of primitive operations executed by an algorithm, as a function of the input size

Algorithm *arrayMax*(A, n)		# operations
$currentMax \leftarrow A[0]$		2
for $i \leftarrow 1$ **to** $n - 1$ **do**		$2 + n$
if $A[i] > currentMax$ **then**		$2(n - 1)$
$currentMax \leftarrow A[i]$		$2(n - 1)$
{ increment counter i }		$2(n - 1)$
return $currentMax$		1
	Total	$7n - 1$

Growth Rate of Running Time

- Changing the hardware/ software environment
  - Affects $\boldsymbol{T}(\boldsymbol{n})$ by a constant factor, but
  - Does not alter the growth rate of $\boldsymbol{T}(\boldsymbol{n})$
- The linear growth rate of the running time $\boldsymbol{T}(\boldsymbol{n})$ is an intrinsic property of algorithm ***arrayMax***
- Growth rates of functions:
  - Linear $\approx \boldsymbol{n}$
  - Quadratic $\approx \boldsymbol{n}^2$
  - Cubic $\approx \boldsymbol{n}^3$
- In a log-log chart, the slope of the line corresponds to the growth rate of the function

Growth rates



The “Order” of Performance: (Big) O

- Basic idea:
  1. Ignore constant factor: computer and language implementation details affect that: go for fundamental rate of increase with problem size.
  2. Consider fastest growing term: Eventually, for large problems, it will dominate.
- Value: Compares fundamental performance difference of algorithms
- Caveat: For smaller problems, big-O worse performer may actually do better

Big-Oh notation

- To simplify the running time estimation,

  for a function *f(n),* we ignore the constants and lower order terms.

  Example: $10n^3+4n^2-4n+5$ is $O(n^3)$

Formally,

Given functions $\boldsymbol{T}(\boldsymbol{n})$ and $\boldsymbol{f}(\boldsymbol{n})$, we say that $\boldsymbol{T}(\boldsymbol{n})$ is $\boldsymbol{O}(\boldsymbol{f}(\boldsymbol{n}))$ if there are positive constants
$\boldsymbol{c}$ and $\boldsymbol{n_0}$ such that

$\boldsymbol{T}(\boldsymbol{n}) \leq \boldsymbol{cf}(\boldsymbol{n})$ for $\boldsymbol{n} \geq \boldsymbol{n_0}$

$T(n) = O(f(n))$ Defined

1. $\exists n_0$ and
2. $\exists c$ such that

If $n > n_0$ then $c \cdot f(n) \geq T(n)$

Example: $T(n) = 3n^2+5n-17$

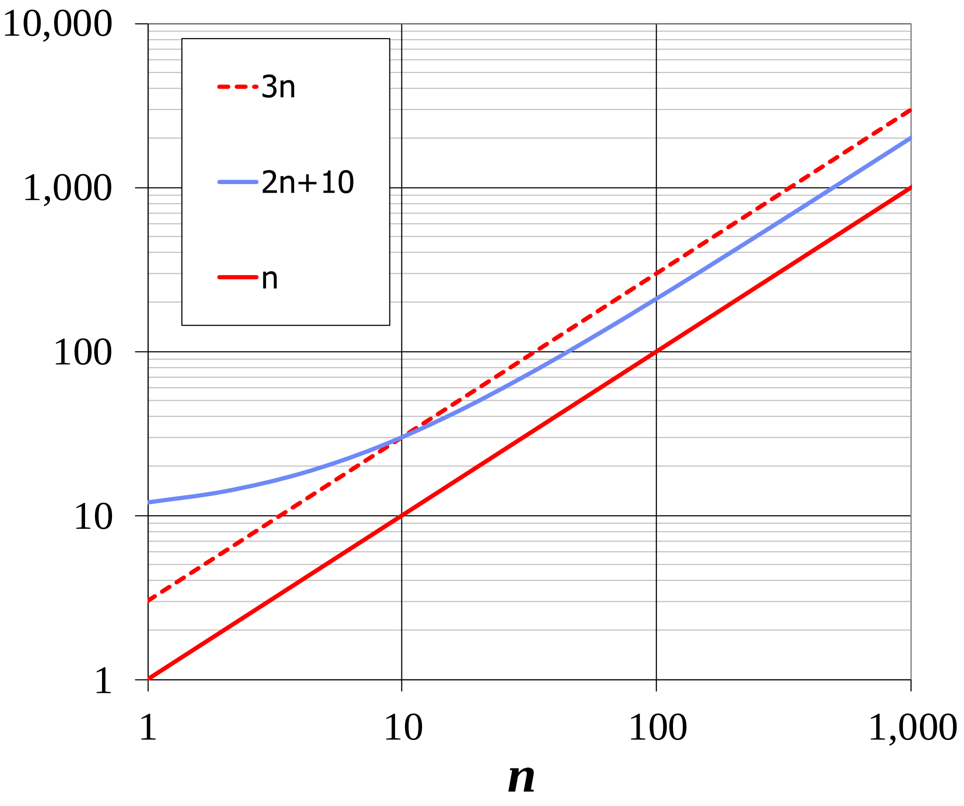
Pick $c = 4$, say; need $4n_0^2 > 3n_0^2+5n_0-17$

$n_0^2 > 5n_0-17$, for which $n_0 = 5$ will do.

$T(n) = O(f(n))$

- $T(n)$ = time for algorithm on input size n
- $f(n)$ = a simpler function that grows at about the same rate

- Example: $T(n) = 3n^2+5n-17$ is $O(n^2)$
  - $f(n)$ has faster growing term
  - no extra leading constant in $f(n)$



Example: $2n + 10$ is $O(n)$

$2n + 10 \leq cn$
$(c - 2)\, n \geq 10$
$n \geq 10/(c - 2)$

Pick $c = 3$ and $n0 = 10$

Big-Oh notation - examples

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

Big-Oh notation - examples

- *7n-2 is O(n)*
  - need c > 0 and $n_0 \geq 1$ such that $7n\text{-}2 \leq c \bullet n$ for $n \geq n_0$
  - this is true for c = 7 and n0 = 1

Big-Oh notation - examples

- *7n-2 is O(n)*
  - need c > 0 and $n_0 \geq 1$ such that $7n\text{-}2 \leq c \bullet n$ for $n \geq n_0$
  - this is true for c = 7 and n0 = 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 $n_0 = 21$

Big-Oh notation - 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 n0 = 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 $n_0 = 2$

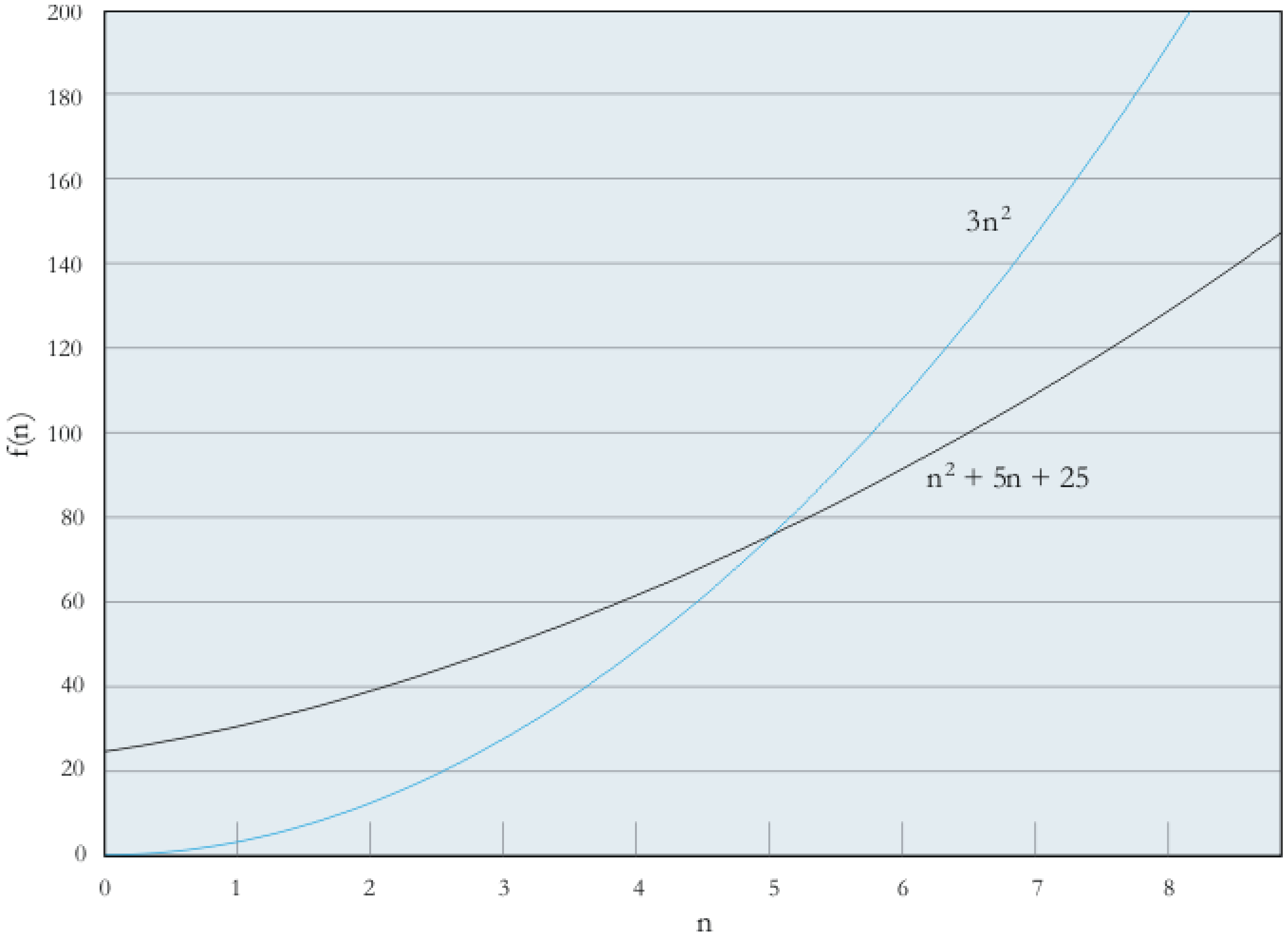
Big-Oh the Upper Bound

- The Big-Oh notation gives an upper bound on the growth rate of a function

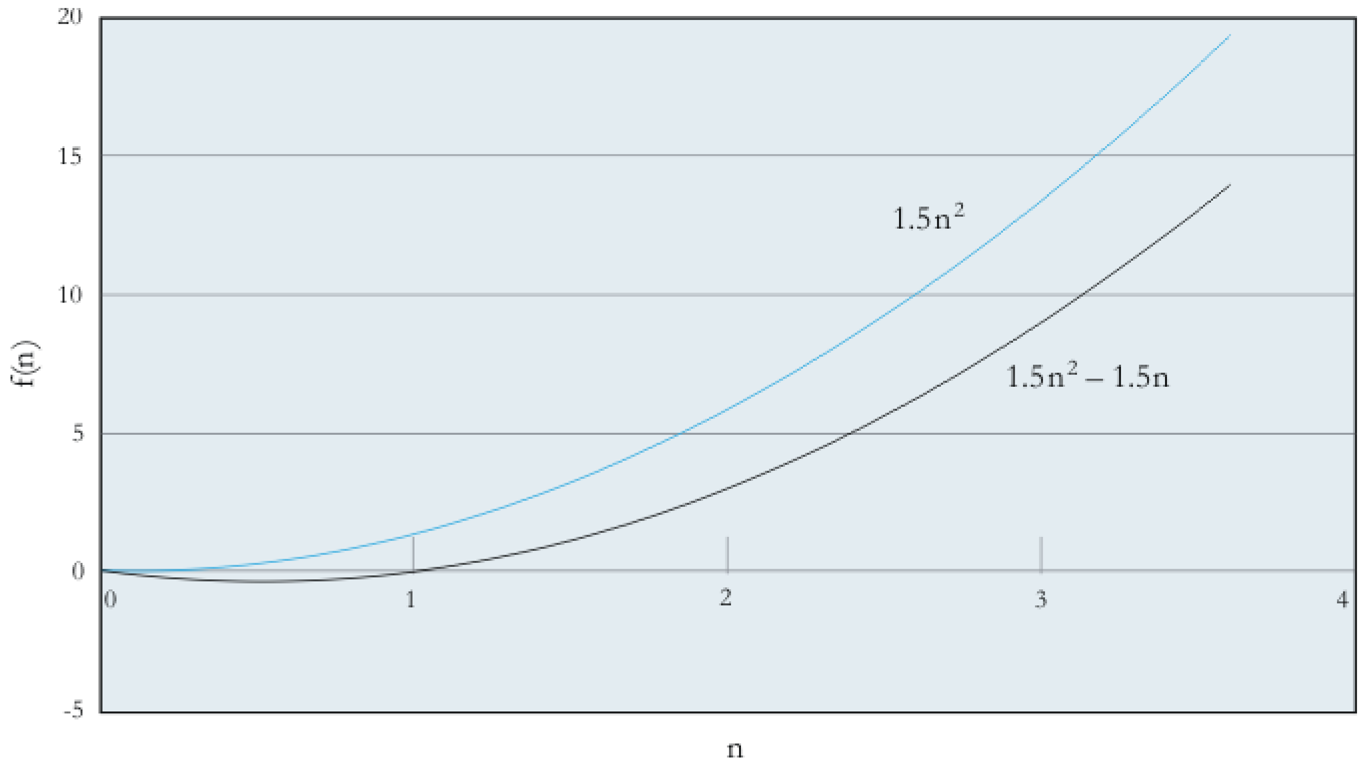
- The statement "$T(n)$ is $O(f(n))$" means that the growth rate of $T(n)$ is no more than the growth rate of $f(n)$

- We can use the big-Oh notation to rank functions according to their growth rate

Efficiency of Algorithms (continued)



Efficiency of Algorithms (continued)



Efficiency of Algorithms (continued)

Symbols used in Quantifying Software Performance

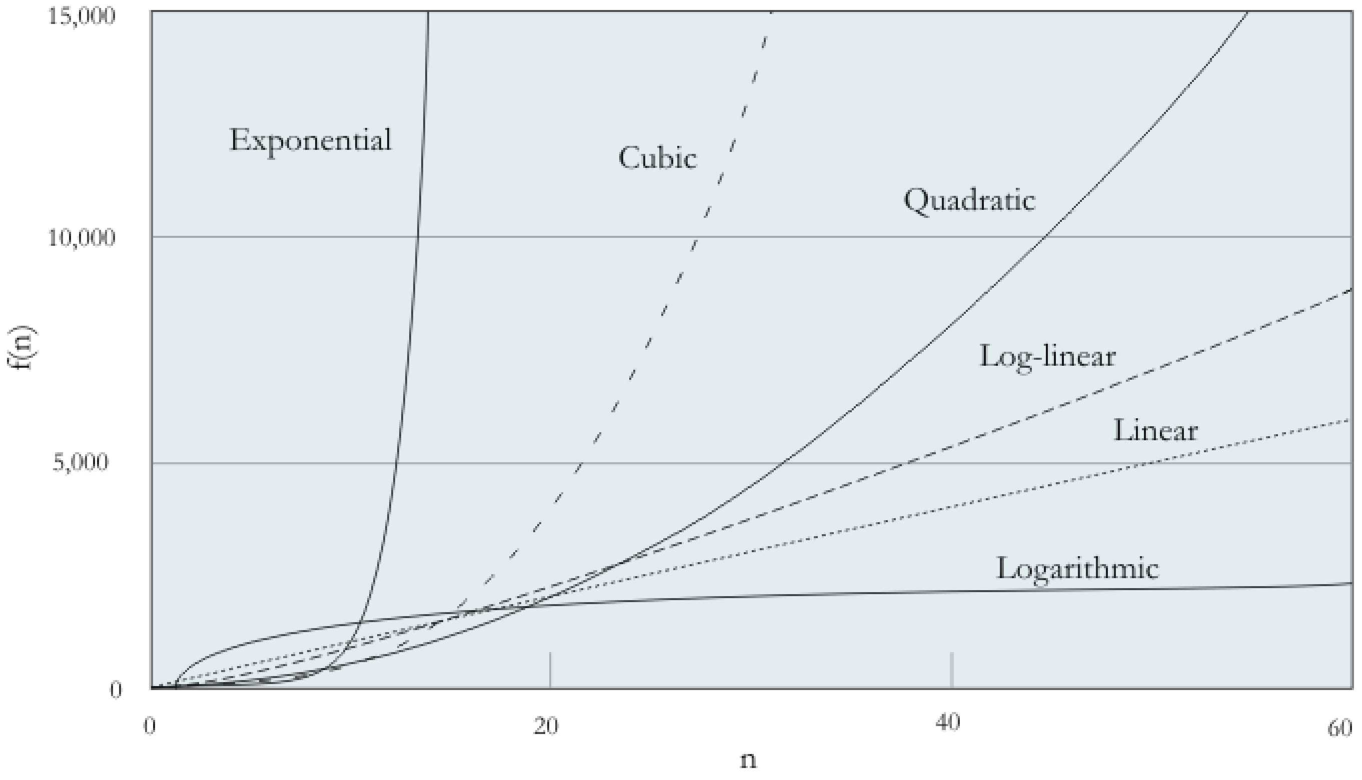
$T(n)$	The time that a function takes as a function of the number of inputs, n. We may not be able to measure or determine this exactly.
$f(n)$	Any function of n. Generaly $f(n)$ will represent a simpler function than $T(n)$, for example n^2 rather than $1.5n^2 - 1.5n$.
$\mathbf{O}(f(n))$	Order of magnitude. $\mathbf{O}(f(n))$ is the set of functions that grow no faster than $f(n)$. We say that $T(n) = \mathbf{O}(f(n))$ to indicate that the growth of $T(n)$ is bounded by the growth of $f(n)$.

Efficiency of Algorithms (continued)

Common Growth Rates

Big-O	Name
O(1)	Constant
O(log n)	Logarithmic
O(n)	Linear
O(n log n)	Log-Linear
O(n^2)	Quadric
O(n^3)	Cubic
O(2^n)	Exponential
O($n!$)	Factorial

Efficiency of Algorithms (continued)



Self check: Which answer?

If the time is approximately doubled when the number of inputs, n, is doubled, then the algorithm grows at a ___________ rate.

A. constant
B. logarithmic
C. linear
D. quadratic

Self check: Which answer?

If the time is approximately doubled when the number of inputs, n, is doubled, then the algorithm grows at a ___________ rate.

A. constant
B. logarithmic
C. linear
D. quadratic

Efficiency Examples

```
int find (int x[], int val) {
  for (int i = 0; i < X_LENGTH; i++) {
    if (x[i] == val)
      return i;
  }
  return -1;  // not found
}
```

What is the time complexity?

Efficiency Examples

```
int find (int x[], int val) {
  for (int i = 0; i < X_LENGTH; i++) {
    if (x[i] == val)
      return i;
  }
  return -1;  // not found
}
```

- Letting n be **`x.length`**:
- Average iterations if *found* =>
  (1+...+n)/n = (n+1)/2 = O(n) iterations
- if *not found* => n = O(n)
- This is called *linear search*.

Efficiency Examples (2)

```
bool all_different (
      int x[], int y[]) {
   for (int i = 0; i < X_LENGTH; i++) {
      if (find(y, x[i]) != -1)
         return false;
   }
   return true;  // no x element found in y
}
```

- Letting m be **X_LENGTH** and n be **Y_LENGTH** m:
- Time if all different = O(m·n) = m · cost of search(n)

Efficiency Examples (3)

```
bool unique (int x[]) {
  for (int i = 0; i < X_LENGTH; i++) {
    for (int j = 0; j < X_LENGTH; j++ {
      if (i != j && x[i] == x[j])
        return false;
    }
  }
  return true;  // no duplicates in x
}
```

Efficiency Examples (3)

```
bool unique (int x[]) {
  for (int i = 0; i < X_LENGTH; i++) {
    for (int j = 0; j < X_LENGTH; j++ {
      if (i != j && x[i] == x[j])
        return false;
    }
  }
  return true;  // no duplicates in x
}
```

- Letting n be `X_LENGTH`:
- Time if unique = n^2 iterations = $O(n^2)$

Efficiency Examples (4)

```
bool unique (int x[]) {
  for (int i = 0; i < X_LENGTH; i++) {
    for (int j = i+1; j < X_LENGTH; j++ {
      if (i != j && x[i] == x[j])
        return false;
    }
  }
  return true;  // no duplicates in x
}
```

- Letting n be **`X_LENGTH`**:
- Time if unique = (n-1)+(n-2)+...+2+1 iterations =
- n(n-1)/2 iterations = O(n^2) *still* ... only **factor of 2 better**

Efficiency Examples (5)

```
for (int i = 1; i < n; i *= 2) {
  do something with x[i]
}
```

What is the time complexity?

Efficiency Examples (5)

```
for (int i = 1; i < n; i *= 2) {
   do something with x[i]
}
```

- Sequence is 1, 2, 4, 8, ..., $\cong$n.
- Number of iterations = $\log_2 n = \log n$.
- Computer scientists generally use base 2 for log, since that matches with number of *bits*, etc.
- Also $O(\log_b n) = O(\log_2 n)$ since chane of base just multiples by a constant: $\log_2 n = \log_b n / \log_b 2$

Chessboard Puzzle

Payment scheme #1: \$1 on first square, \$2 on second, \$3 on third, ..., \$64 on 64^{th}.

Payment scheme #2: 1¢ on first square, 2¢ on second, 4¢ on third, 8¢ on fourth, etc.

Which is best?

Chessboard Puzzle Analyzed

Payment scheme #1: Total = \$1+\$2+\$3+...+\$64 = $\$64\times65/2$ = \$1755

Payment scheme #2: $1¢+2¢+4¢+...+2^{63}¢ = 2^{64}-1¢$ = \$184.467440737 *trillion*

- ❑Many cryptographic schemes require $O(2^n)$ work to break a key of length n bits.
- ❑A key of length n=40 is perhaps breakable,
- ❑but a key with length n=256 is NOT TODAY!!!!!