Performance Analysis

CS212:Data Structure

Mathematical Foundations

The Constant Function:

$$f(n) = c$$

For some fixed constant c, such as $c = 5$, $c = 100$ *or* $c = 1000{,}000$.

- Regardless of the value of n, $f(n)$ will always be equal to the constant value c.
- The most fundamental constant function is:

$$g(n) = 1$$

- Any constant function $f(n) = c$, can be written as a constant c times $g(n)$. That is:

$$f(n) = cg(n)$$

Mathematical Foundations

The Logarithm Function:

$$f(n) = \log_b n \text{. for some constant } b > 1$$

$$x = \log_b n \text{ if and only if } b^x = n$$

- The value b is known as the ***base*** of the logarithm. The most common base for the logarithm function in computer sciences is ***2***. Therefore, it is typical not to write the base when it is 2:

$$\log n = \log_2 n$$

- **properties of logarithms:**

$$\log_b(xy) = \log_b x + \log_b y$$
$$\log_b (x/y) = \log_b x - \log_b y$$
$$\text{Log}_b x^a = a\log_b x$$
$$\text{Log}_b a = \log_x a / \log_x b$$

Mathematical Foundations

The linear Function:

$$f(n) = n$$

The N-Log-N Function:

$$f(n) = n \log n$$

The Quadratic Function:

$$f(n) = n^2$$

The Cubic Function:

$$f(n) = n^3$$

- All the function listed so far are part of a larger class of functions called the ***polynomial.***

$$f(n) = a_0 + a_1 n + a_2 n^2 + a_3 n^3 + \ldots + a_d n^d$$

Where $a_0, a_1, a_2, \ldots, a_d$ are constants, called ***coefficients*** of the polynomial, and $a_d \neq 0$. the integer *d,* which indicates the highest power in the polynomial, is called the ***degree*** of the polynomial.

Mathematical Foundations

The Exponential Function:

$$f(n) = b^n$$

where b is a positive constant, called the ***base***, and the argument n is the ***exponent***.

- **properties of exponentials:**

$a^{(b+c)} = a^b a^c$

$a^{bc} = (a^b)^c$

$a^b / a^c = a^{(b-c)}$

$b = a^{\log_a b}$

$b^c = a^{c*\log_a b}$

Mathematical Foundations

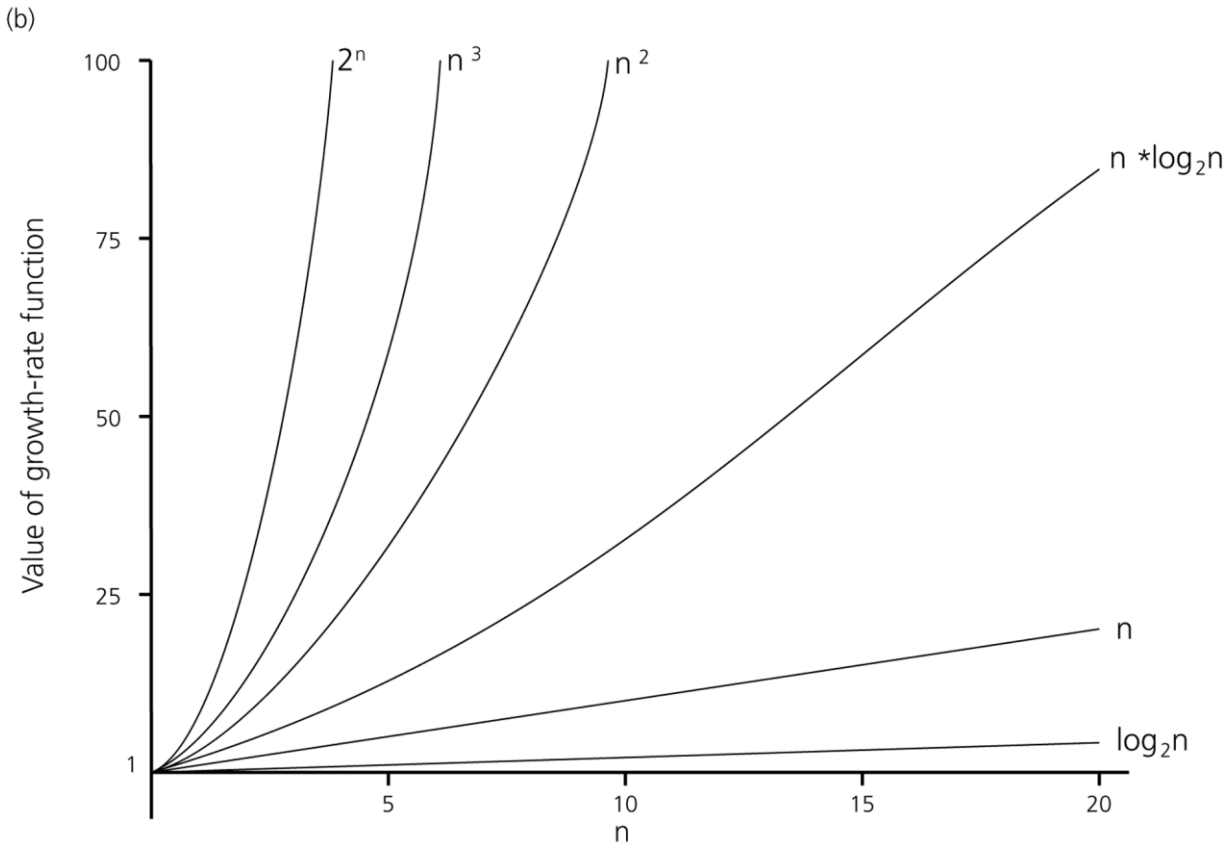
Comparing Growth Rate in Tabular form:

(a)

Function	n = 10	100	1,000	10,000	100,000	1,000,000
1	1	1	1	1	1	1
$\log_2 n$	3	6	9	13	16	19
n	10	10^2	10^3	10^4	10^5	10^6
$n * \log_2 n$	30	664	9,965	10^5	10^6	10^7
n^2	10^2	10^4	10^6	10^8	10^{10}	10^{12}
n^3	10^3	10^6	10^9	10^{12}	10^{15}	10^{18}
2^n	10^3	10^{30}	10^{301}	$10^{3,010}$	$10^{30,103}$	$10^{301,030}$

Mathematical Foundations

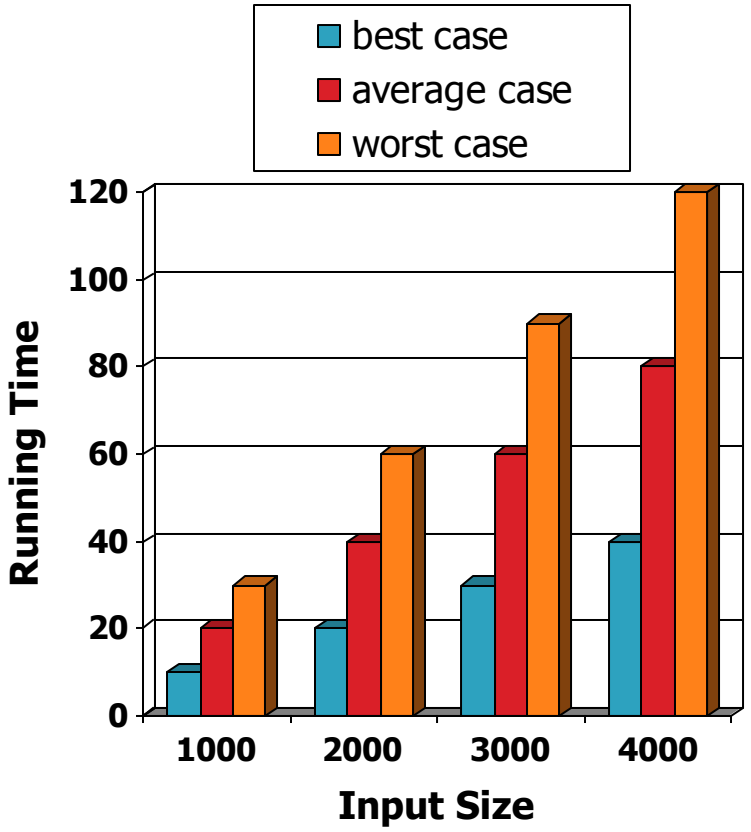
Comparing Growth Rate in graphical form:



Analysis of Algorithms

Running Time:

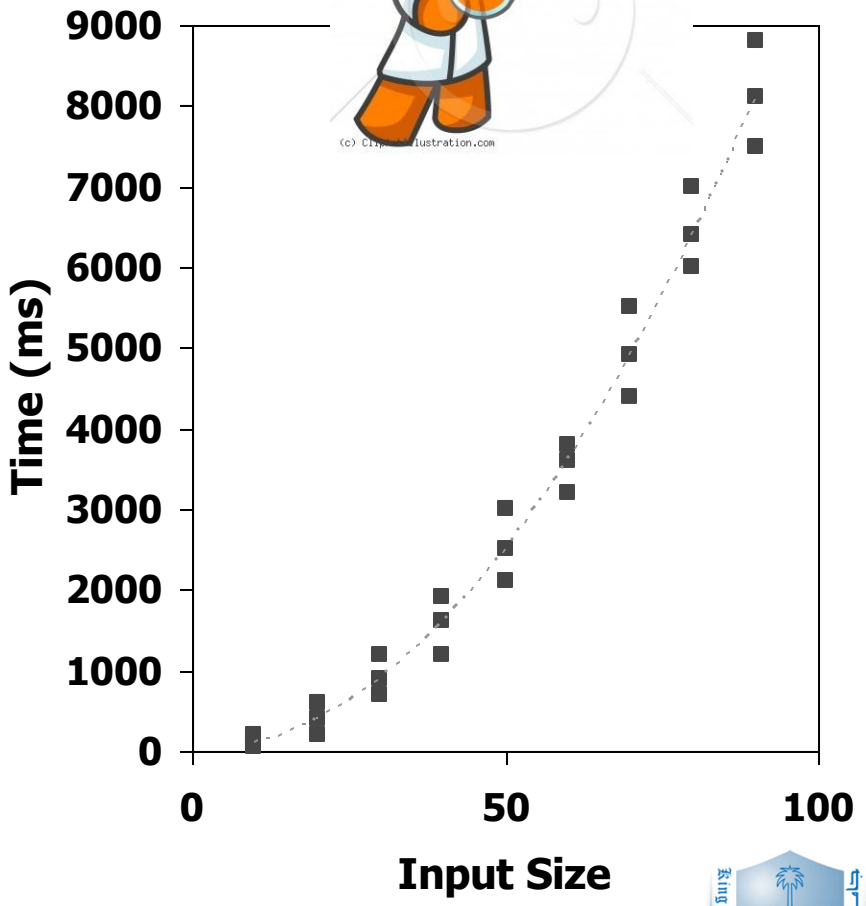
- Most algorithms transform input objects into output objects.
- The running time of an algorithm typically grows with the input size.
- Average case time is often difficult to determine.
- We focus on the worst case running time.
  - Easier to analyze
  - Crucial to applications such as games, finance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



Analysis of Algorithms

Experimental Studies:

- Write a program implementing the algorithm.
- Run the program with inputs of varying size and composition.
- Use a method like System.currentTimeMillis() to get an accurate measure of the actual running time.
- Plot the results



Analysis of Algorithms

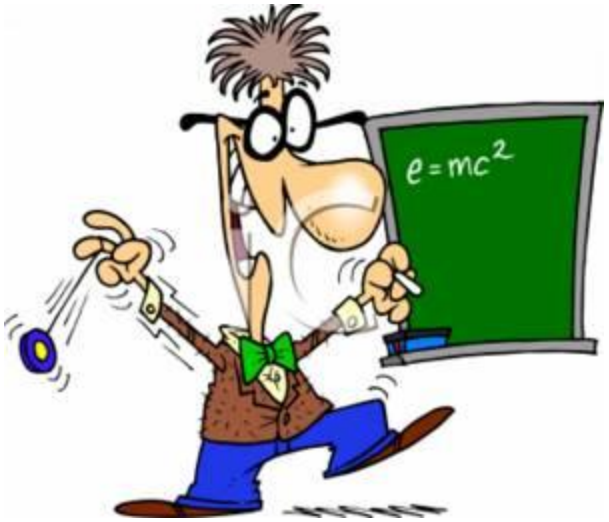
Limitations of Experiments:

- 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.

Analysis of Algorithms

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.



Analysis of Algorithms

Pseudocode:

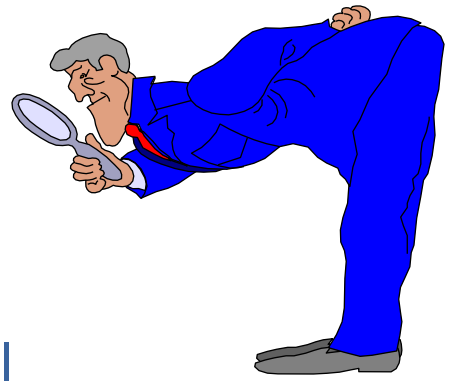
- High-level description of an algorithm.
- More structured than English prose.
- Less detailed than a program.
- Preferred notation for describing algorithms.
- Hides program design issues.

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
```

Analysis of Algorithms



Pseudocode Details:

- Control flow
  - **if** … **then** … [**else** …]
  - **while** … **do** …
  - **repeat** … **until** …
  - **for** … **do** …
  - Indentation replaces braces
- Method declaration

  Algorithm ***method*** (***arg*** [, ***arg***…])

  Input …

  Output …
- Method call

  var.method (***arg*** [, ***arg***…])
- Return value

  return ***expression***
- Expressions

  ← Assignment (like = in Java)

  = Equality testing (like == in Java)

  n^2 Superscripts and other mathematical formatting allowed

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 execution time.

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

Counting Primitive Operations

- Primitive operation corresponds to a low-level instruction with a constant execution time.
- Instead of determine the specific execution time of each primitive operation, simply ***count*** how many primitive operation are executed.
- This operation count will correlate to an actual running time in a specific computer.
- By inspecting the pseudocode, we can determine the maximum number of primitive operations executed by an algorithm, as a function of the input size.

Counting Primitive Operations

	Statements	S/E	Freq.	Total
1	Algorithm Sum1(a[],n)			
2	{			
3	S = 0.0;			
4	for i=1 to n do			
5	s = s+a[i];			
6	return s;			
7	}			

Counting Primitive Operations

	Statements	S/E	Freq.	Total
1	Algorithm Sum1(a[],n)	0	-	0
2	{	0	-	0
3	S = 0.0;	1	1	1
4	for i=1 to n do	1	n+1	n+1
5	s = s+a[i];	1	n	n
6	return s;	1	1	1
7	}	0	-	0
				2n+3

n+1

n

Counting Primitive Operations

	Statements	S/E	Freq.	Total
1	Algorithm Sum2(a[],n,m)			
2	{			
3	for i=1 to n do;			
4	for j=1 to m do			
5	s = s+a[i][j];			
6	return s;			
7	}			

Counting Primitive Operations

	Statements	S/E	Freq.	Total
1	Algorithm Sum2(a[],n,m)	0	-	0
2	{	0	-	0
3	for i=1 to n do (n+1)	1	n+1	n+1
4	for j=1 to m do (m+1)	1	n(m+1)	n(m+1)
5	s = s+a[i][j]; (n, m)	1	nm	nm
6	return s;	1	1	1
7	}	0	-	0
				2nm+2n+2

Estimating Running Time

- Algorithm ***Sum1*** executes $2n + 3$ 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 ***Sum1.*** Then

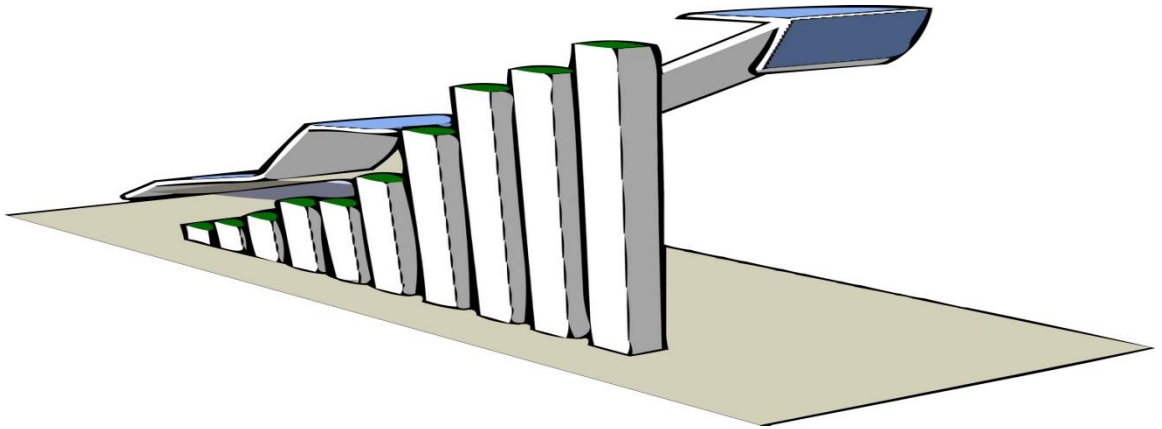
$$a(2n + 3) \leq T(n) \leq b(2n + 3)$$

- 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 ***Sum1.***

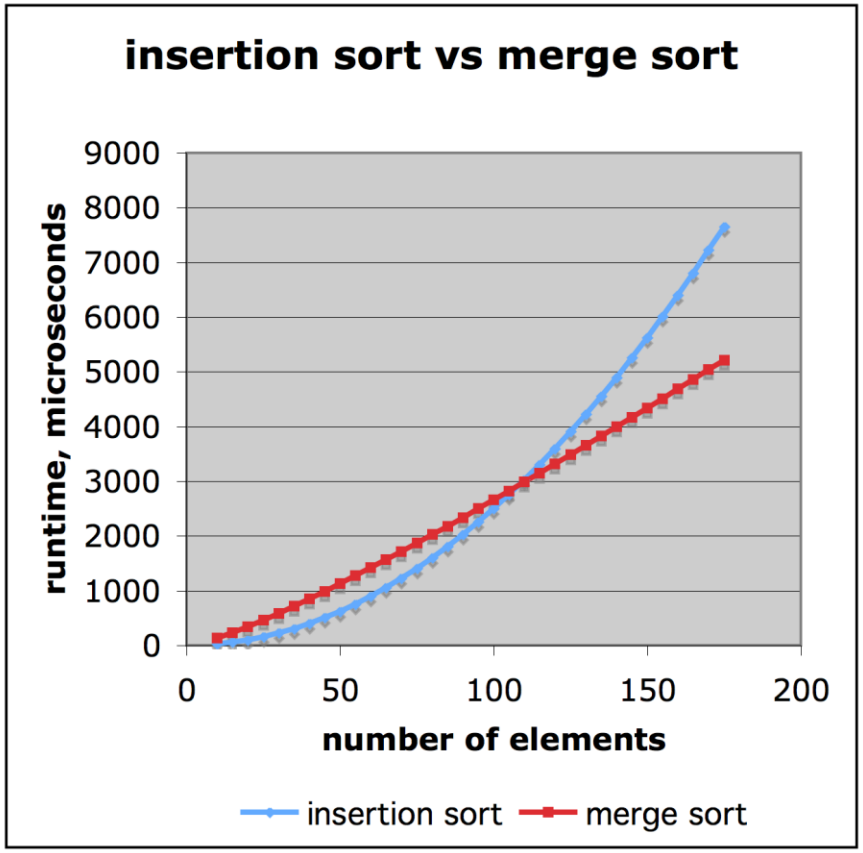


Why Growth Rate Matters

if runtime is...	time for n + 1	time for 2 n	time for 4 n
c log n	c log (n + 1)	c (log n + 1)	c(log n + 2)
c n	c (n + 1)	2c n	4c n
c n log n	~ c n log n + c n	2c n log n + 2cn	4c n log n + 4cn
c n^2	~ c n^2 + 2c n	**4c n^2**	16c n^2
c n^3	~ 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

Comparison of Two Algorithms (an example)



insertion sort is
$n^2 / 4$

merge sort is
$2 n \log n$

sort a million items using a basic PC?

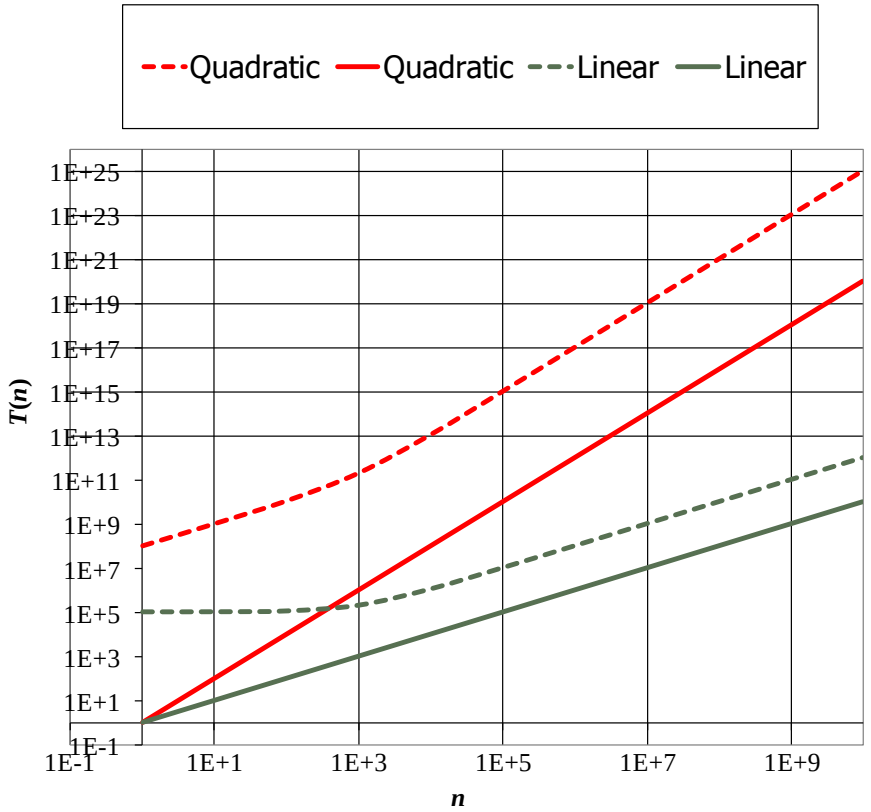
insertion sort takes
roughly 70 hours

while

merge sort takes
roughly 40 seconds

Constant Factors

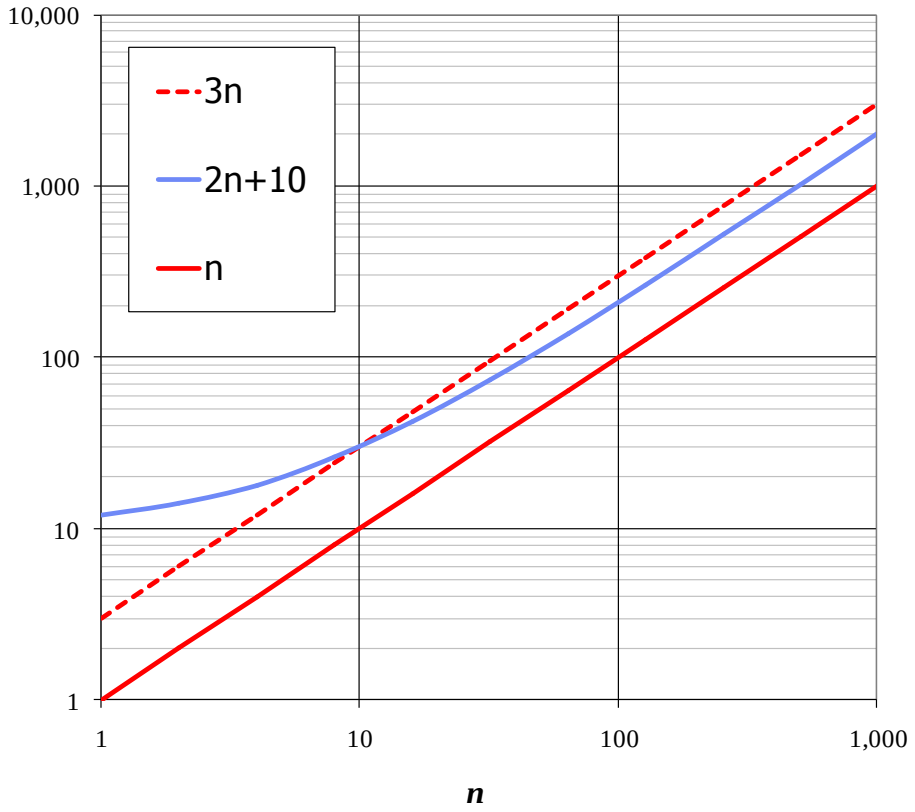
- The growth rate is not affected by
  - constant factors or
  - lower-order terms
- Examples
  - $10^2 n + 10^5$ is a linear function
  - $10^5 n^2 + 10^8 n$ is a quadratic function



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

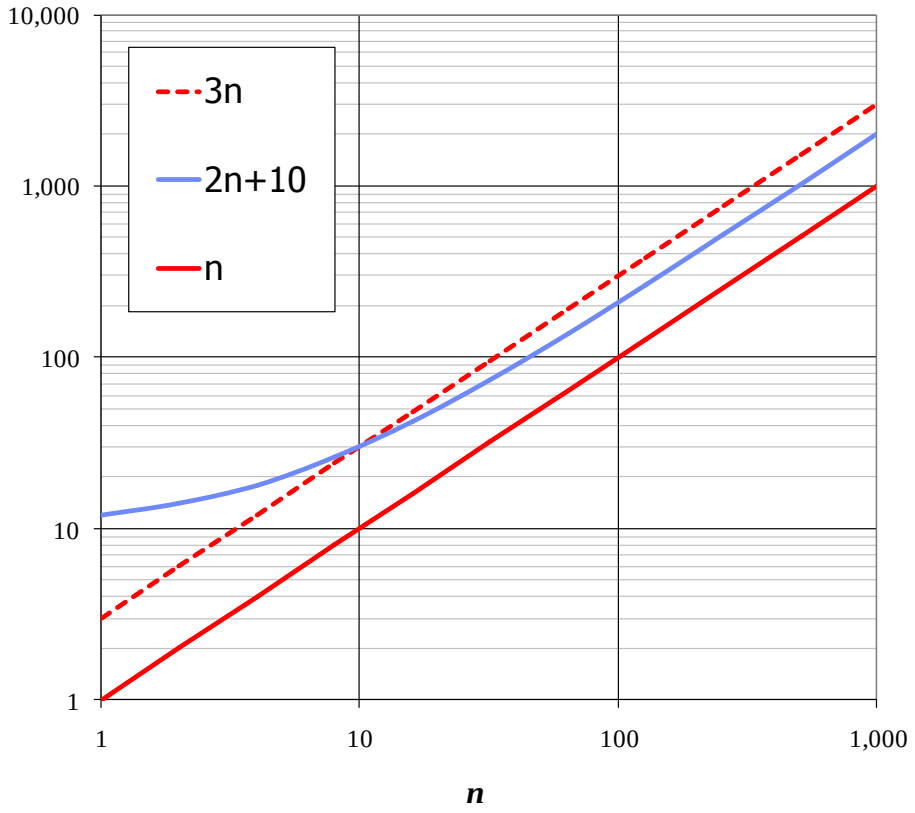
  $f(n) \leq cg(n)$ for $n \geq n_0$
- Example: $2n + 10$ is $O(n)$
  - $2n + 10 \leq cn$
  - $10 \leq \mathbf{cn} - 2\mathbf{n}$
  - $10 \leq (c - 2)\, n$
  - $10/(c - 2) \leq n$
  - Pick $c = 3$ and $n_0 = 10$



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

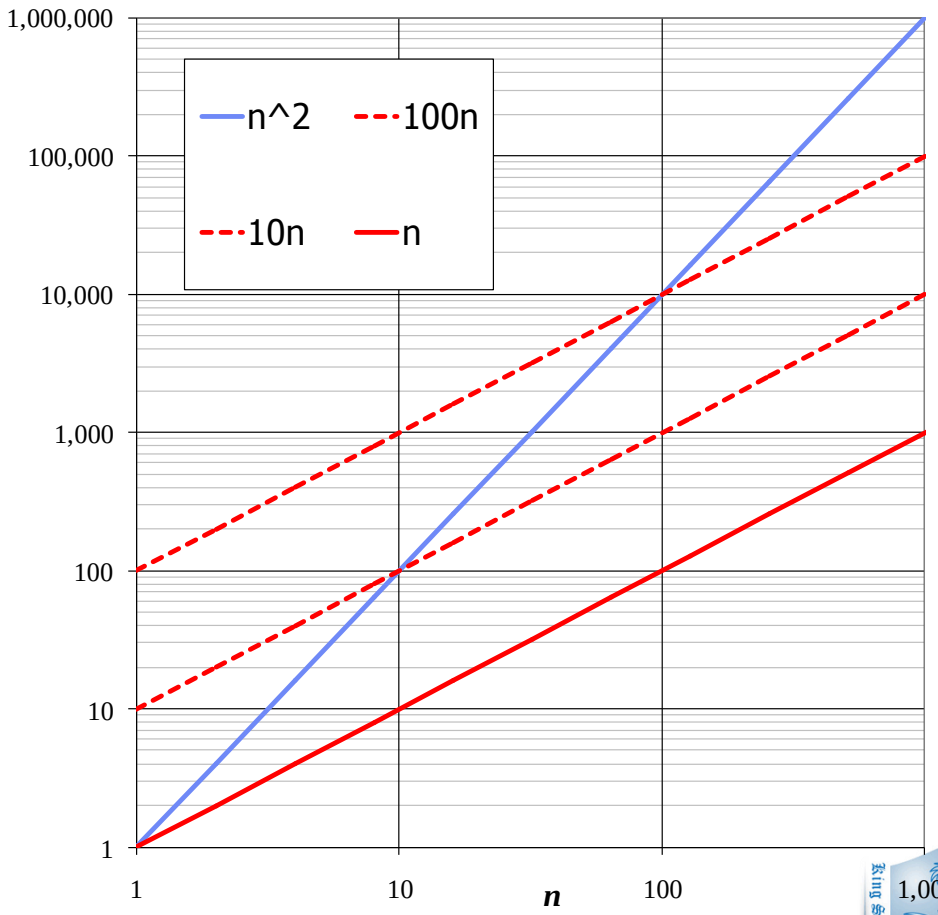
  $f(n) \leq cg(n)$ for $n \geq n_0$
- Example: $2n + 10$ is $O(n)$
  - $2n + 10 \leq cn$
  - $10 \leq \mathbf{cn} - 2\mathbf{n}$
  - $10 \leq (c - 2)\, n$
  - $10/(c - 2) \leq n$
  - Pick $c = 3$ and $n_0 = 10$



$$2n + 10 \leq 2n + 10n = 12n$$
$$n_0 = 1$$
$$c = 12$$

Big-Oh Example

- 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



More Big-Oh Examples

- **7n-2**

  7n-2 is O(n)

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

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

More Big-Oh Examples

- **7n-2**

  7n-2 is O(n)

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

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

$$7n - 2 \leq 7n$$
$$n_0 = 1$$
$$c = 7$$

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

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

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

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

More Big-Oh Examples

- **7n-2**

  7n-2 is O(n)

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

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

$$3n^3 + 20n^2 + 5 \leq 3n^3 + 20n^3 + 5n^3 = 28n^3$$
$$n_0 = 1$$
$$c = 28$$

- **3 log n + 5**

  3 log n + 5 is O(log n)

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

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

More Big-Oh Examples

- **$7n-2$**

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

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

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

$$3 \log n + 5 \leq 3 \log n + 5 \log n = 8 \log n$$
$$n_0 = 2$$
$$c = 8$$

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.

Big-Oh Rules

- If is $f(n)$ a polynomial of degree d, then $f(n)$ is $O(n^d)$, i.e.,
  1. Drop lower-order terms
  2. Drop constant factors

- Use the smallest possible class of functions
  Say "$2n$ is $O(n)$" instead of "$2n$ is $O(n^2)$".

- Use the simplest expression of the class
  Say "$3n + 5$ is $O(n)$" instead of "$3n + 5$ is $O(3n)$"

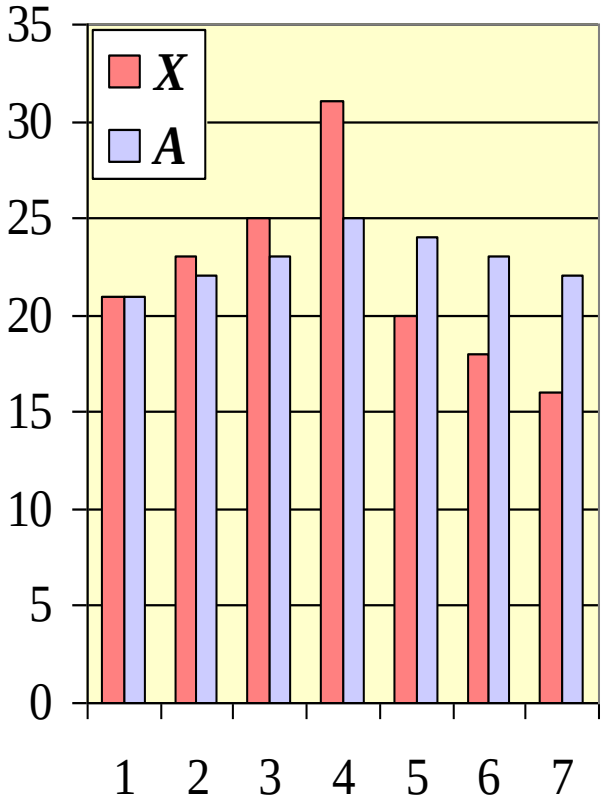
Asymptotic Algorithm Analysis

- 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 determine that algorithm ***Sum1*** executes at most $2n + 3$ primitive operations
  - We say that algorithm ***Sum1*** "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 (an example)

- 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



Computing Prefix Averages (an example)

- The following algorithm computes prefix averages in quadratic time.

	Statements	S/E	Freq.	Total
1	**Algorithm** ***prefixAverages1***(***X, n***)	0	-	0
2	{	0	-	0
3	***A*** ← new array of ***n*** integers	1	n	n
4	**for** ***i*** ← 0 **to** ***n*** – 1 **do**	1	n+1	n+1
5	***s*** ← ***X***[0]	1	n	n
6	**for** ***j*** ← 1 **to** ***i*** **do**	1	1+2+3+… (n-1)	n(n+1)/2
7	***s*** ← ***s*** + ***X***[***j***]	1	1+2+3+… (n-1)	n(n+1)/2
8	***A***[***i***] ← ***s*** / (***i*** + 1)	1	n	n
9	**return** ***A***	1	1	1
10	}	0	-	0
				$2n^2+6n+2$

Thus, Algorithm ***prefixAverages1*** is $O(n^2)$.

Computing Prefix Averages (an example)

$$\sum_{k=1}^{n} k = \frac{n(n+1)}{2},$$

- The following algorithm computes prefix averages in quadratic time.

	Statements	S/E	Freq.	Total
1	**Algorithm** ***prefixAverages1***(***X, n***)	0	-	0
2	{	0	-	0
3	***A*** ← new array of ***n*** integers	1	n	n
4	**for** ***i*** ← 0 **to** ***n*** – 1 **do**	1	n+1	n+1
5	***s*** ← ***X***[0]	1	n	n
6	**for** ***j*** ← 1 **to** ***i*** **do**	1	1+2+3+… (n-1)	n(n+1)/2
7	$\sum_{k=1}^{n} k$ ***s*** ← ***s*** + ***X***[***j***]	1	1+2+3+… (n-1)	n(n+1)/2
8	***A***[***i***] ← ***s*** / (***i*** + 1)	1	n	n
9	**return** ***A***	1	1	1
10	}	0	-	0
				$2n^2+6n+2$

n

Thus, Algorithm ***prefixAverages1*** is $O(n^2)$.

Computing Prefix Averages (an example)

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

	Statements	S/E	Freq.	Total
1	**Algorithm** ***prefixAverages2***(***X, n***)	0	-	0
2	{	0	-	0
3	***A*** ← new array of ***n*** integers	1	n	n
4	***s*** ← ***0***	1	1	1
5	**for** ***i*** ← 0 **to** ***n*** – 1 **do**	1	n+1	n+1
6	***s*** ← ***s*** + ***X[i]***	1	n	n
7	***A[i]*** ← ***s*** / (***i*** + 1)	1	n	n
8	**return** ***A***	1	1	1
9	}	0	-	0
				4n+3

Thus, Algorithm ***prefixAverages2*** is ***O(n)***.

Computing Prefix Averages (an example)

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

	Statements	S/E	Freq.	Total
1	**Algorithm *prefixAverages2*(*X*, *n*)**	0	-	0
2	{	0	-	0
3	*A* ← new array of *n* integers	1	n	n
4	*s* ← *0*	1	1	1
5	**for** *i* ← 0 **to** *n* – 1 **do**	1	n+1	n+1
6	n *s* ← *s* + *X[i]*	1	n	n
7	*A*[*i*] ← *s* / (*i* + 1)	1	n	n
8	**return** *A*	1	1	1
9	}	0	-	0
				4n+3

Thus, Algorithm ***prefixAverages2*** is ***O*(*n*)**.

Big-Oh From Smallest to Largest

$O(1)$		Constant
$O(\log n)$		Logarithmic
$O(n)$		Linear
$O(n \log n)$		n log n
$O(n^c)$	$O(n^2)$, $O(n^3)$, $O(n^{16})$, etc	Polynomial
$O(c^n)$	$O(1.6^n)$, $O(2^n)$, $O(3^n)$, etc	Exponential
$O(n!)$		Factorial
$O(n^n)$		n power n

Big-Oh Examples

$O(1)$	Push, Pop, Enqueue (if there is a tail reference), Dequeue, Accessing an array element
$O(\log n)$	Binary search
$O(n)$	Linear search
$O(n \log n)$	Heap sort, Quick sort (average), Merge sort
$O(n^2)$	Selection sort, Insertion sort, Bubble sort
$O(n^3)$	Matrix multiplication
$O(2^n)$	Towers of Hanoi
$O(n!)$	All permutation of N elements
$O(n^n)$	….

Revision

- Each countable step is weighted as 1, anything else is weighted 0 **(S/E)**
- Count the time each step is executed. This can be (1) time, constant time (5, 10, 21,…) or variable time (n, m, n+1, n^2,…) **(Freq)**
- Multiply **(S/E)** by **(Freq)** to get **(Total)**
- Having done all the above for each step, sum **(Total)** for each step togather to get the complexity

Revision

- **for/while/do-while** are examples where there are repetition/frequency.
  - **for/while, <, ++ loops**
    - Freq = max – initial
  - **for/while, <=, ++ loops**
    - Freq = max – initial + 1
  - **for/while, >, -- loops**
    - Freq = initial – max
  - **for/while, >=, -- loops**
    - Freq = initial – max + 1

- **IMPORTANT: for/while** checking step/line should add +1 (for last check). Internal loop steps use the above formulas.

Revision

- **do-while, <, ++ loops**
  - Freq = 1+ max – initial
- **do-while, <=, ++ loops**
  - Freq = 1 + max – initial + 1
- **do-while, >, -- loops**
  - Freq = 1+ initial – max
- **do-while, >=, -- loops**
  - Freq = 1 + initial – max + 1

- **IMPORTANT:** both **do-while** checking step/line and internal loop steps use the above formulas.

Revision

- **In other word:**
  - ++
    - max – initial
  - --
    - initial – max
  - <=, >=
    - add (+1)
  - If, **do-while**
    - add (+1), for both checking and internal steps
  - Else (**for/while**)
    - add (+1) for checking step only.

- **IMPORTANT:** This only apply for incrementing/decrementing loops with simple checking (<,>,<=,>=). These are generalization (there are **many exceptions**)

Revision

- **One exception example:**

```
for (int i = 0; i < n; i++)
    for(int j = 0; j < i; j++)
        System.out.println(j)
```

  - In this case, the internal loop depends on the external loop (j < i). Therefore, the number of loops is changing each time.
    - The checking line will be executed 1+2+3+…+n
    - The *println* will be executed 0+1+2+…+(n-1)
  - This is an example/approximation for $\sum_{k=1}^{n} k = \frac{n(n+1)}{2},$

- Such cases require careful counting

Revision

- Big-Oh for a function f(n)
  - Drop lower-order terms
  - Drop constant factors
  - Use the smallest possible class
  - Use the simplest expression of the class

- Or, Find highest order term, and drop everything else (including constants)

Revision

- **Proving f(n) is O(g(n))**
  - f(n) is O(g(n)) if there are positive constants c and n_0 such that:

    $f(n) \leq cg(n)$ for $n \geq n_0$

- Or:
  - Drop negative terms
  - Upgrade lower-order terms to the same level of the highest-order term level
  - Work it from there to calculate c and n_0

Excercises

a).

```
s = 0;
for (i = 1; i < n-1; i++)
  s = s + 1;
```

b).

```
s = 0;
for (i = n-5; i > 4; i--)
  s = s + i;
```

Excercises

a).

```
s = 0;                              1
for (i = 1; i < n-1; i++)           n-1
  s = s + 1;                        n-2
```

Total: 2n-2 **O(n)**

b).

```
s = 0;
for (i = n-5; i > 4; i--)
  s = s + i;
```

Excercises

a).

```
s = 0;                              1
for (i = 1; i < n-1; i++)           n-1
  s = s + 1;                        n-2
```

Total: 2n-2 O(n)

b).

```
s = 0;                              1
for (i = n-5; i > 4; i--)           n-8
  s = s + i;                        n-9
```

Total: 2n-16 O(n)

Excercises

c).

```
s = 0;
for (i = 1; i <= n; i++)
  for (j = 1; j <= n; j++)
     s = s + 1;
```

d).

```
i = 0;
while (i <= 10)
  i = i + 1;
```

Excercises

c).

```
s = 0;                                  1
for (i = 1; i <= n; i++)                n+1
  for (j = 1; j <= n; j++)              n(n+1)
    s = s + 1;                          n²
```

Total: $2n^2+2n+2$ **$O(n^2)$**

d).

```
i = 0;
while (i <= 10)
  i = i + 1;
```

Excercises

c).

```
s = 0;                              1
for (i = 1; i <= n; i++)            n+1
  for (j = 1; j <= n; j++)          n(n+1)
    s = s + 1;                      n²
```

Total: $2n^2+2n+2$ **$O(n^2)$**

d).

```
i = 0;                              1
while (i <= 10)                     12
  i = i + 1;                        11
```

Total: 24 **O(1)**

Excercises

e).

```
s = 0;
for (i = 1; i <= n; i++)
  for (j = 0; j <= n; j++)
    for (k = 0; k <= n; k++)
          s = s + 1;
```

Excercises

e).

```
s = 0;                                  1
for (i = 1; i <= n; i++)                n+1
  for (j = 0; j <= n; j++)              n(n+2)
    for (k = 0; k <= n; k++)            n(n+1)(n+2)
          s = s + 1;                    n(n+1)(n+1)
```

Total: $O(n^3)$

Excercises

f).

```
s = 0;
for (i = 0; i <= n; i++)
  for (j = 0; j < i; j++)
     s = s + 1;
```

g).

```
s = 0;
for (i = 0; i <= n; i++)
  for (j = i+1; j <= n; j++)
     s = s + 1;
```

Excercises

f).

```
s = 0;                                  1
for (i = 0; i <= n; i++)                n+2
  for (j = 0; j < i; j++)               1+2+...+(n+1)      ~n(n+1)/2
    s = s + 1;                          1+2+...+n          =n(n+1)/2
```

Total: $O(n^2)$

g).

```
s = 0;
for (i = 0; i <= n; i++)
  for (j = i+1; j <= n; j++)
    s = s + 1;
```

Excercises

f).

```
s = 0;                                  1
for (i = 0; i <= n; i++)                n+2
  for (j = 0; j < i; j++)               1+2+...+(n+1)      ~n(n+1)/2
    s = s + 1;                          1+2+...+n          =n(n+1)/2
```

Total: $O(n^2)$

g).

```
s = 0;                                  1
for (i = 0; i <= n; i++)                n+2
  for (j = i+1; j <= n; j++)            (n+1)+n+...+2+1    ~n(n+1)/2
    s = s + 1;                          n+(n-1)+...+2+1    =n(n+1)/2
```

Total: $O(n^2)$