PERFORMANCE ANALYSIS

CS212: Data Structure

Outline

- Performance analysis
  - space complexity
  - time complexity
  - By experiment
  - By analysis
- Growth rate of a function
- Big O notation

Introduction

- We saw two implementations of the ADT List, a linked implementation and an array implementation.
  - The question that we want to answer now: which implementation gives a better performance?
- In general, when having different algorithms that solve the same problem, how to compare their performances? Which one is **better**?

- we need to define what we mean by **best**.
  - time complexity—the time it takes to execute
  - space complexity—the memory it needs to execute.

Introduction

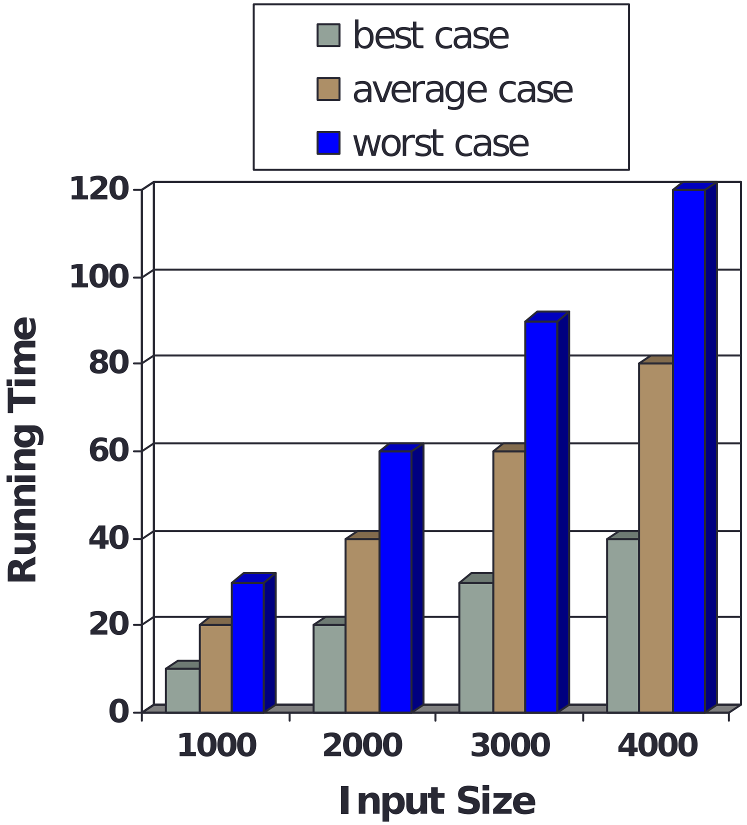
Performance analysis :

- The process of measuring the complexity of algorithms
  - We will concentrate on the time complexity of algorithms,

- There are two ways to compare algorithms:
  - **Experimental analysis** : compare the running time for different input sizes.
  - **Theoretical analysis** : analyze the algorithms independently of the implementation (hardware/software).

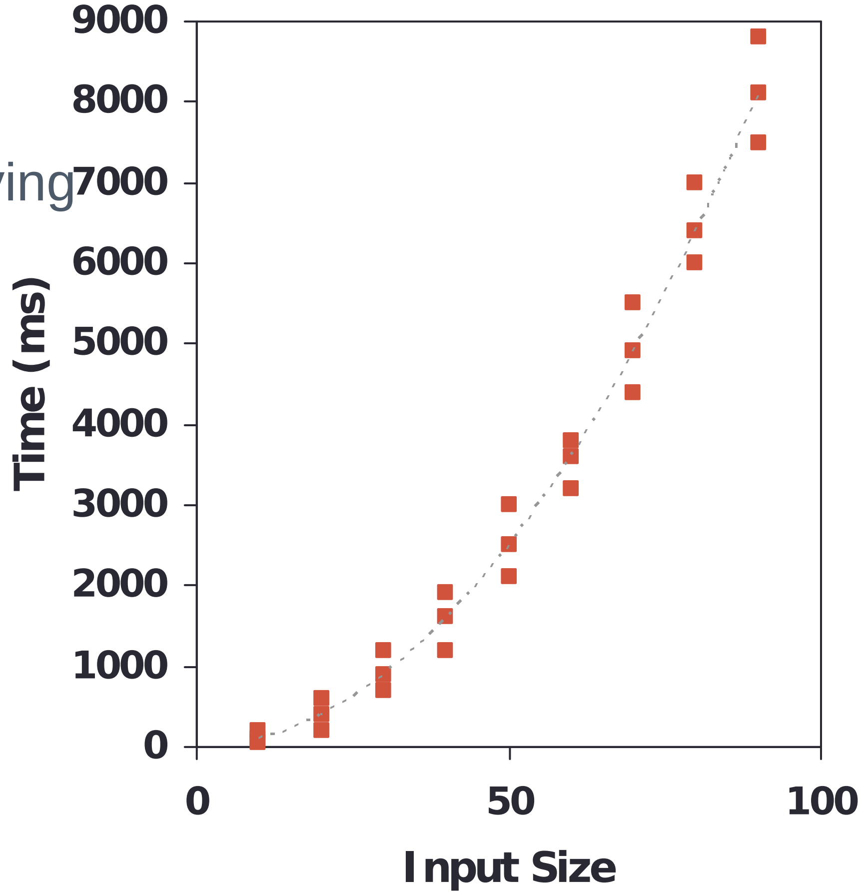
Experimental analysis

- 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 analysis

- 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



Experimental analysis

Limitations:

- 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 (depends on them).

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.

Theoretical Analysis

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***

Theoretical Analysis

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

Theoretical Analysis

Counting Primitive Operations

- Primitive operation corresponds to a low-level instruction with 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

- 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.

Theoretical Analysis

Counting Primitive Operations

	Statements	Step/ Exec utio	Freq .	Total
1	Algorithm Sum1(a[],n)	0	-	-
2	{	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	-	-

Theoretical Analysis

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

$\boxed{2n+3}$

Theoretical Analysis

Counting Primitive Operations

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

Theoretical Analysis

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	1	n+1	n+1
4	for j=1 to m do	1	n(m+1)	n(m+1)
5	s = s+a[i][j];	1	nm	nm
6	return s;	1	1	1
7	}	0	-	0

$\boxed{2nm+2n+2}$

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

$$a\,(2n + 3) \le T(n) \le b(2n + 3)$$

- 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
$c\ n^2$	$\sim c\ n^2 + 2c\ n$	**$4c\ n^2$**	$16c\ n^2$
$c\ n^3$	$\sim c\ n^3 + 3c\ n^2$	$8c\ n^3$	$64c\ n^3$
$c\ 2^n$	$c\ 2^{n+1}$	$c\ 2^{2n}$	$c\ 2^{4n}$

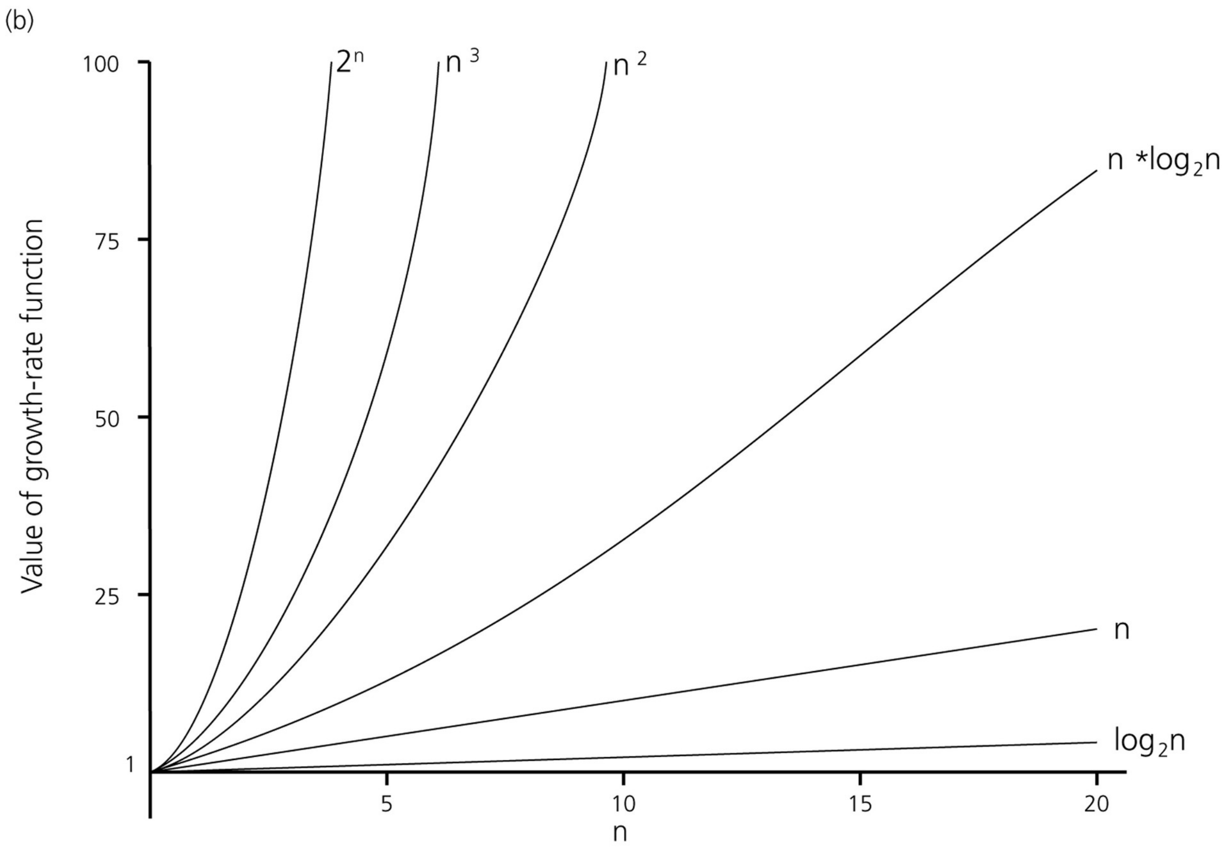
runtime quadruples when problem size doubles

Comparing Growth Rate in Tabular form

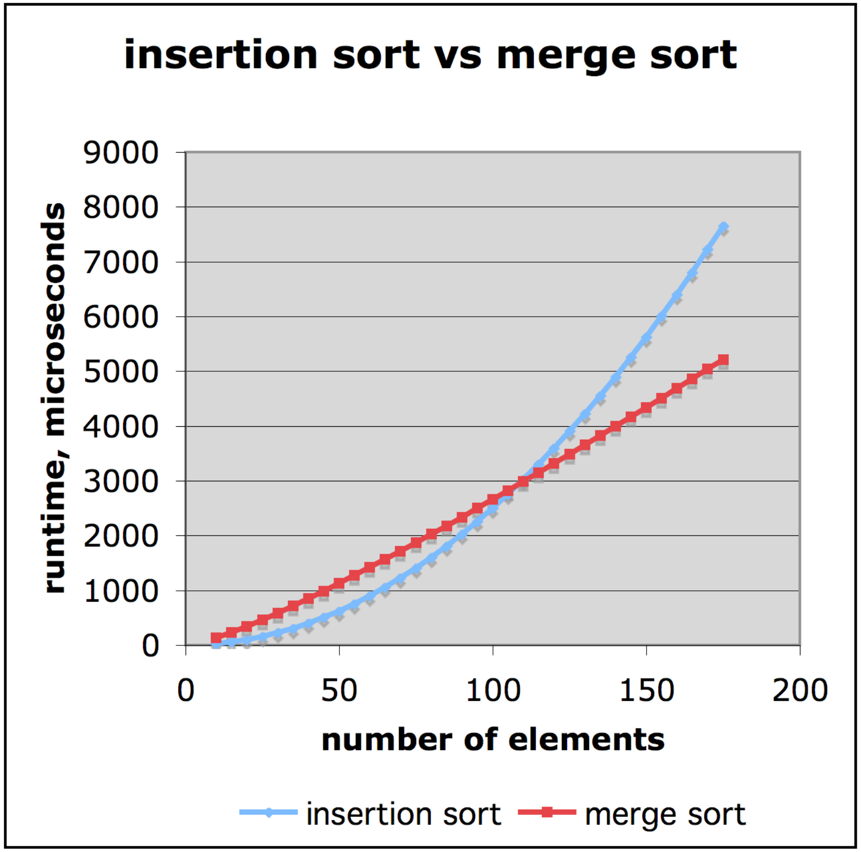
(a)

Function	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}	$10^{3,010}$	$10^{30,103}$	$10^{301,030}$

Comparing Growth Rate in Tabular form



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?

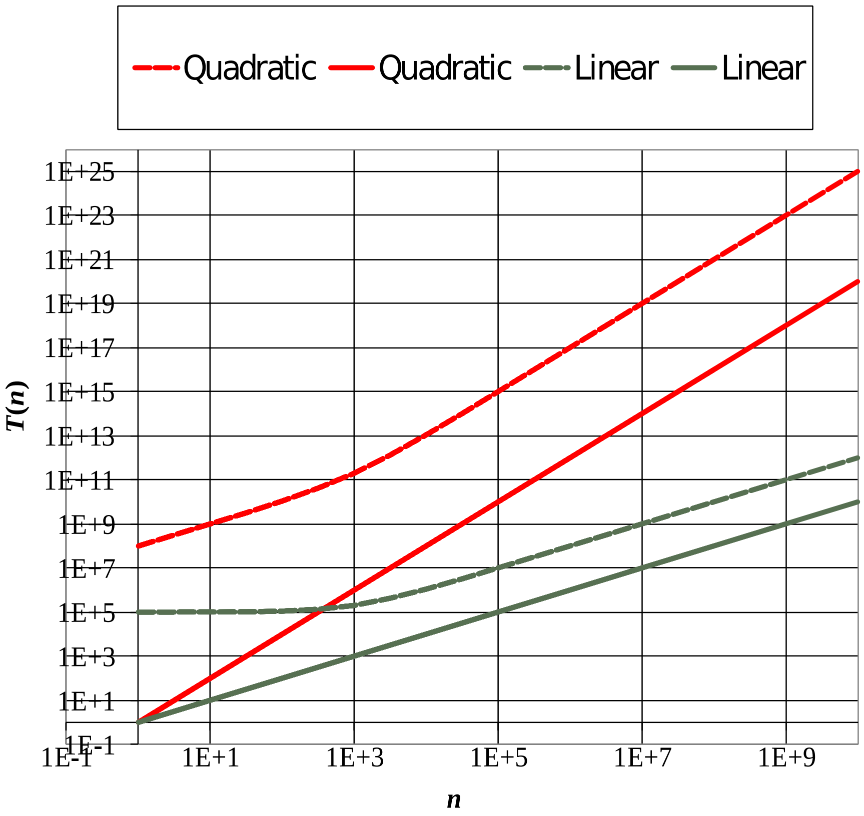
insertion sort takes roughly 70 hours

while

merge sort takes roughly 40 seconds

Constant Factors

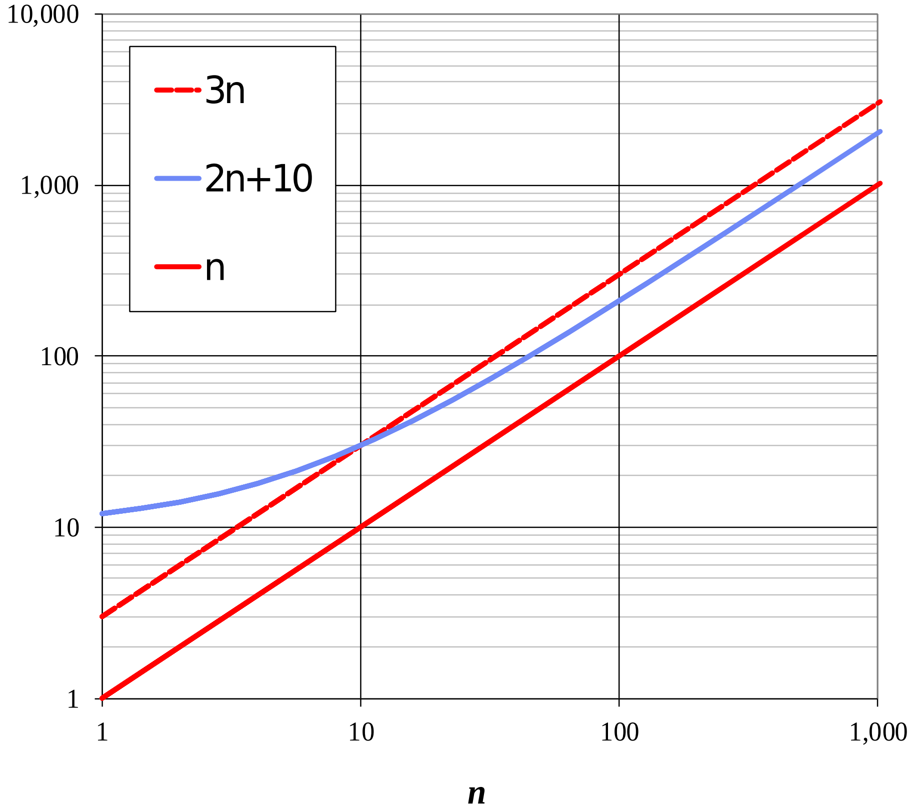
- The growth rate is not affected by
  - constant factors or
  - lower-order terms
- Examples
  - $10^2 \boldsymbol{n} + 10^5$ is a linear function
  - $10^5 \boldsymbol{n}^2 + 10^8 \boldsymbol{n}$ is a quadratic function



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

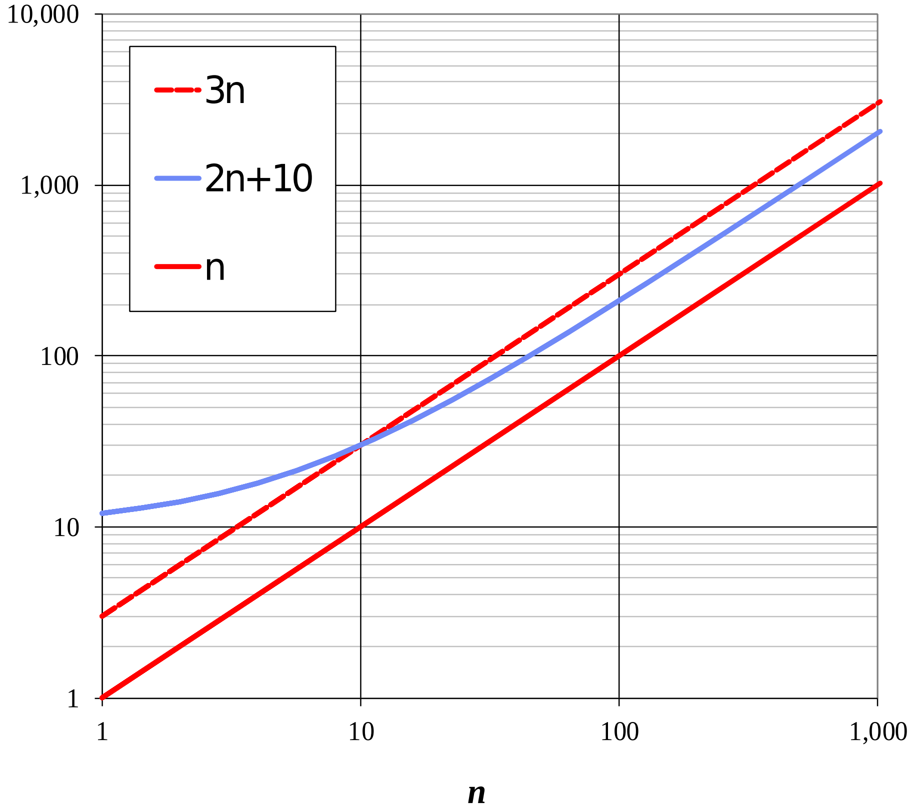
  $f(n) \leq cg(n)$ for $n \geq n_0$
- Example: $2n + 10$ is $O(n)$
  - $2n + 10 \leq cn$
  - $10 \leq cn - 2n$
  - $10 \leq (c - 2) n$
  - $10/(c - 2) \leq n$
  - Pick $c = 3$ and $n_0 = 10$



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 $O(n)$
  - $2n + 10 \leq cn$
  - $10 \leq cn - 2n$
  - $10 \leq (c - 2)\, n$
  - $10/(c - 2) \leq n$
  - Pick $c = 3$ and $n_0 = 10$



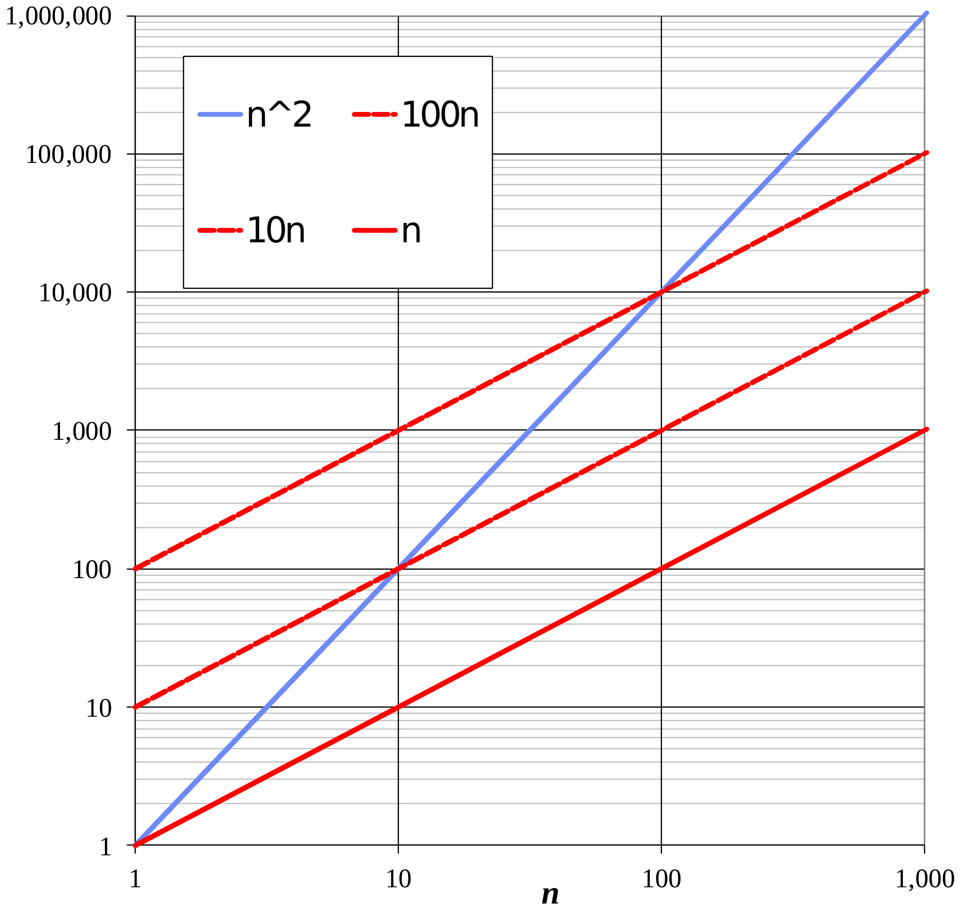
$2n + 10 \leq 2n + 10n = 12n$

$n_0 = 1$

$c = 12$

Big-Oh Example

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



More Big-Oh Examples

- **7n-2** $7n-2 \le 7n$ $f(n) \le c\,g(n)$

  $n \le n_0$

  $n \ge n_0$; $n \ge 1$ $c > 0$

  $c = 7$

  7n-2 is O(n)

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

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

- **$3n^3 + 20n^2 + 5$** $\le (3+20+5)n^3$

  $3n^3 + 20n^3 + 5n^3$ $c = 28$

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

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

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

- **3 log n + 5** $\le 8 \log n$ $c = 8$

  $n_0 = 2$

  3 log n + 5 is O(log n)

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

  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 $n_0 = 1$

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

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

  $n_0 = 1$

  $c = 28$

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

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

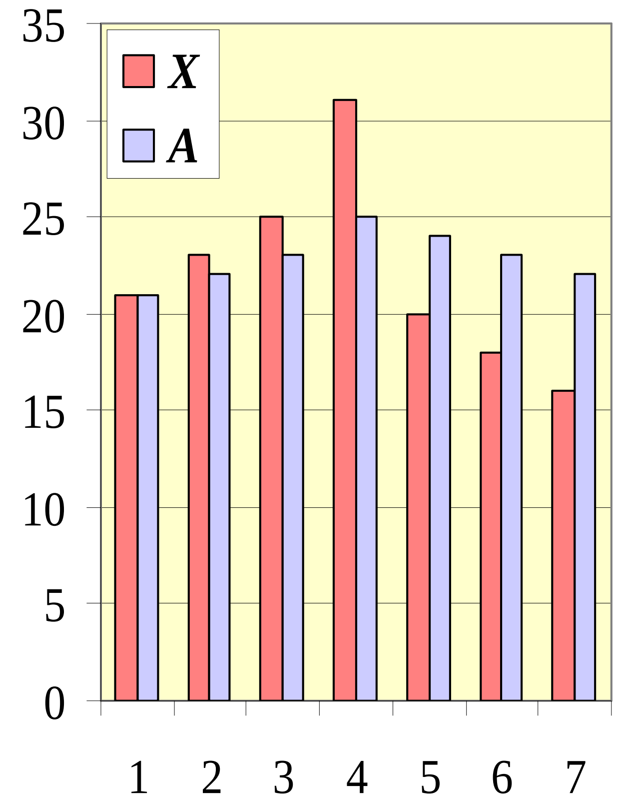
Computing Prefix Averages (an example)

- 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***:

  $\boldsymbol{A}[\boldsymbol{i}] = (\boldsymbol{X}[0] + \boldsymbol{X}[1] + \ldots + \boldsymbol{X}[\boldsymbol{i}])/(i+1)$

- 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***) x	0	-	0
2	{ x	0	-	0
3	$A \leftarrow$ new array of ***n*** integers ✓	1	n → stores n elements	n
4	**for** ***i*** $\leftarrow$ 0 **to** ***n*** - 1 **do** ✓ n → n+1 checking steps	1	n+1	n+1
5	***s*** $\leftarrow$ ***X***[0] → n times	1	n	n
6	**for** ***j*** $\leftarrow$ 1 **to** ***i*** **do** $1+2+3+4+\dots(n) = \frac{n(n+1)}{2}$	1	1+2+3+… (n)	n(n+1)/2
7	***s*** $\leftarrow$ ***s*** + ***X***[***j***] $1+2+3+4\dots(n-1) = \frac{n(n-1)}{2}$	1	1+2+3+… (n-1)	n(n-1)/2
8	***A***[***i***] $\leftarrow$ ***s*** / (***i*** + 1) → n times	1	n	n
9	**return** ***A*** → 1	1	1	1
10	}	0	-	0
				n^2+4n+2

n $\frac{n(n-1)}{2}$

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	**n** **for** ***j*** ← 1 **to** ***i*** **do**	1	1+2+3+… (n)	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
				n^2+4n+2

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 → n	1	n	n
4	***s*** ← ***0*** → 1	1	1	1
5	**for** ***i*** ← 0 **to** ***n*** - 1 **do** → n+1	1	n+1	n+1
6	***s*** ← ***s*** + ***X[i]*** → n	1	n	n
7	***A[i]*** ← ***s*** / (***i*** + 1) → n	1	n	n
8	**return** ***A*** → 1	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	***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 → speeds up the process by 2
$O(n)$	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 together 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)
```

$n+1$

$\frac{n(n+1)}{2}$

$\frac{n(n-1)}{2}$

- 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)

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

- This is an example/approximation for

- 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:

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

- 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++) $n-1-1+1 \rightarrow n-1$

  s = s + 1; $\rightarrow n-2$

$O(n)$

b).

s = 0;

$n-5-4+1$

for (i = n-5; i > 4; i--) $n-8$

  s = s + i; $n-9$

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

n n

d).

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

O(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; **n^2**

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

d).

i = 0; $\rightsquigarrow 1$

while (i <= 10) $\rightsquigarrow 10-0+1+1=12$

i = i + 1; $\rightarrow 11 \cdot \underline{n}^0 \rightarrow 1$

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; **n^2**

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; ~ 1

n
for (i = 1; i <= n; i++) ~ n-1+1+1 = n+1

n+1
for (j = 0; j <= n; j++) (n+2)n

n+1
for (k = 0; k <= n; k++) (n+1)(n)(n+2)

s = s + 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;
```

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;
n+1  for (i = 0; i <= n; i++)   n+2
       for (j = 0; j < i; j++)   1 2 3 . . . n → n(n+1)/2
       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;

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

for (j = i+1; j <= n; j++) 1 . . . . n-1 + n —— n+1 $\frac{n(n+1)}{2}$

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: $O(n^2)$