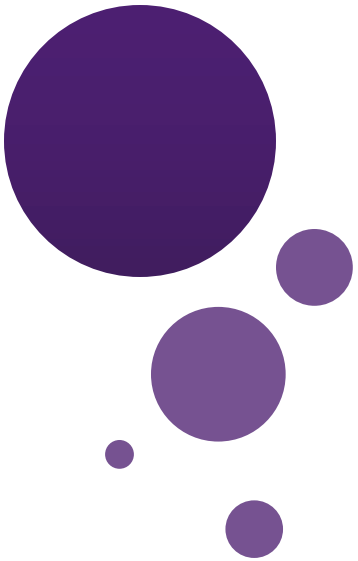


Lecture 4: Algorithm, Growth Function and Complexity, and Integer Division

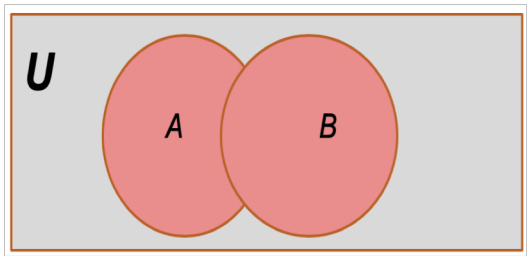
Dr. Chengjiang Long
Computer Vision Researcher at Kitware Inc.
Adjunct Professor at SUNY at Albany.
Email: **clong2@albany.edu**

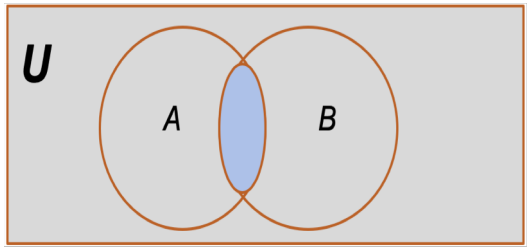
Office Hours

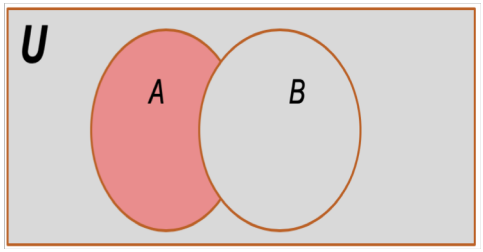
- Instructor's Office Hour (by appointment)
  - Wed 12:45 PM – 3:45 PM at UAB 412E.
- TA's Office Hours
  - Oguz Aranay, Wed 9:30 AM – 11:00 AM at UAB 401.

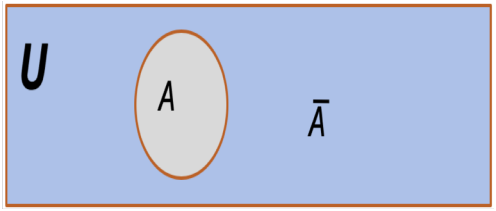
Recap Previous Lecture

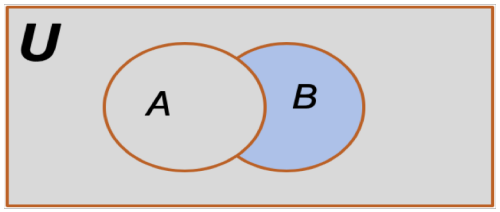
- Set (definition, set builder, operators, cardinality)
- Function (definition, property, inverse function, composition)

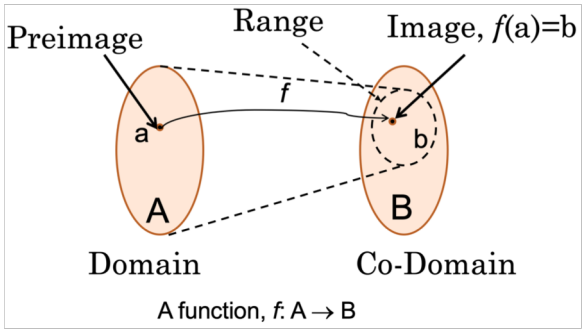


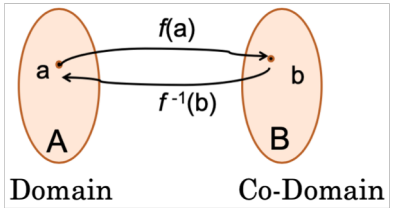


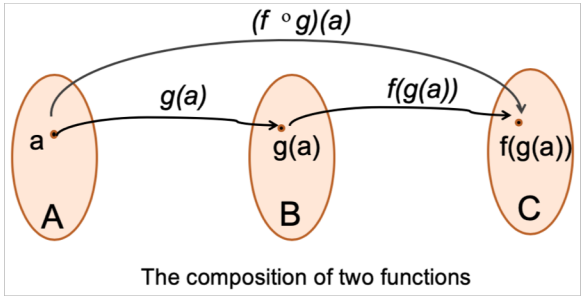






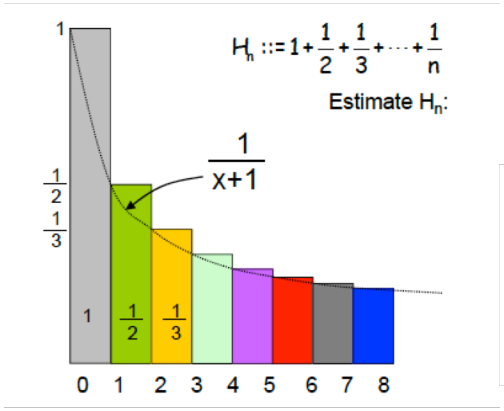






Recap Previous Lecture

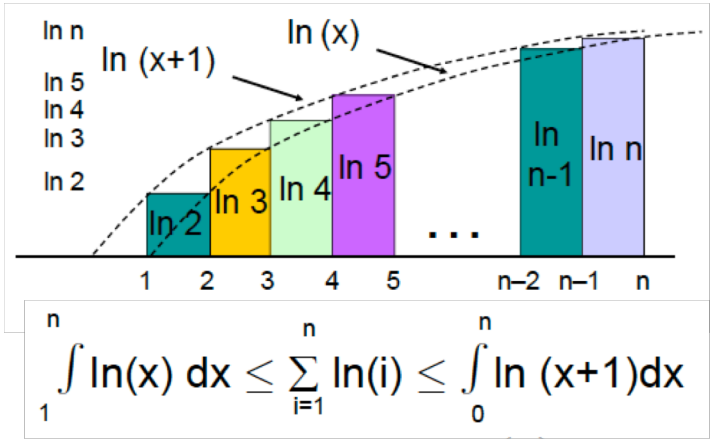
- Sequences (arithmetic, geometric, harmonic, summation)
- Matrices (matrix operation, zero-one matrices, Boolean product of zero-one product)



$$\int_0^n \frac{1}{x+1}dx \le 1+\frac{1}{2}+\frac{1}{3}+\ldots+\frac{1}{n}$$

$$\int_1^{n+1} \frac{1}{x}dx \le H_n$$

$$\ln(n+1) \le H_n$$



$$\int_1^n \ln(x)\,dx \le \sum_{i=1}^{n} \ln(i) \le \int_0^n \ln(x+1)dx$$

$$\mathbf{A} = \begin{bmatrix} 1 & 0 \\ 0 & 1 \\ 1 & 0 \end{bmatrix}, \qquad \mathbf{B} = \begin{bmatrix} 1 & 1 & 0 \\ 0 & 1 & 1 \end{bmatrix}.$$

$$\mathbf{A} \odot \mathbf{B} = \begin{bmatrix} (1 \wedge 1) \vee (0 \wedge 0) & (1 \wedge 1) \vee (0 \wedge 1) & (1 \wedge 0) \vee (0 \wedge 1) \\ (0 \wedge 1) \vee (1 \wedge 0) & (0 \wedge 1) \vee (1 \wedge 1) & (0 \wedge 0) \vee (1 \wedge 1) \\ (1 \wedge 1) \vee (0 \wedge 0) & (1 \wedge 1) \vee (0 \wedge 1) & (1 \wedge 0) \vee (0 \wedge 1) \end{bmatrix}$$

Outline

- Algorithm
- Growth Function and Complexity
- Integers and Division
- Modular Arithmetic

Algorithm

Algorithms

- When presented a problem, e.g., given a sequence of integers, find the larges one
- Construct a model that translates the problem into a mathematical context
  - Discrete structures in such models include sets, sequences, functions, graphs, relations, etc.
- A method is needed that will solve the problem (using a sequence of steps)
- **Algorithm**: a sequence of steps

Algorithm

- **Algorithm**: a finite set of precise instructions for performing a computation or for solving a problem

- Some problems have no solution.
- Some people know solution for some problem.
- Some solutions for some problems could be described as a sequence of instructions.
- Some sequences of instructions are **algorithms**.

Algorithm

- Example: describe an algorithm for finding the maximum (largest) value in a finite sequence of integers

Example

- Perform the following steps
  - Set up temporary maximum equal to the first integer in the sequence
  - Compare the next integer in the sequence to the temporary maximum, and if it is larger than the temporary maximum, set the temporary maximum equal to this
  - Repeat the previous step if there are more integers in the sequence
  - Stop when there are no integers left in the sequence. The temporary maximum at this point is the largest integer in the sequence.

Pseudo code

- Provide an intermediate step between English and real implementation using a particular programming language

  procedure *max*(a_1, a_2, …, a_n: integers)

  max := a_1

  for i:=*2* **to** *n*

  if *max* < a_i **then** *max*:=a_i

  {*max* is the largest element}

Prosperities of algorithm

- **Input**: input values from a specified set
- **Output**: for each set of input values, an algorithm produces output value from a specified set
- **Definiteness**: steps must be defined precisely
- **Correctness**: should produce the correct output values for each set of input values
- **Finiteness**: should produce the desired output after a finite number of steps
- **Effectiveness**: must be possible to perform each step exactly and in a finite amount of time
- **Generality**: applicable for all problems of the desired form, not just a particular set of input values

Algorithmic Problems

As we know not all of the problems have algorithmic solution. Those that have are called algorithmic problem.

Prove that programming languages may be used to solve only algorithmic problem.

The very common application of algorithms:

- Searching Problems: finding the position of a particular element in a list.
- Sorting problems: putting the elements of a list into increasing order.
- Optimization Problems: determining the optimal value (maximum or minimum) of a particular quantity over all possible inputs.

Searching Problems

- The general *searching problem* is to locate an element x in the list of distinct elements $a_1, a_2, \ldots, a_n$, or determine that it is not in the list.
- The solution to a searching problem is the location of the term in the list that equals x (that is, i is the solution if $x = a_i$) or 0 if x is not in the list.

Linear Search

procedure *linear search*(x:integer, a_1, a_2, …, a_n: distinct integers)

i := 1

while ($i \leq n$ **and** $x \neq a_i$)

    i:=i+1

if $i < n$ **then** *location*:=n

else *location:=0*

{*location* is the index of the term equal to x, or is 0 if x is not found}

Binary search

- Given a sorted list, by comparing the element to be located to the middle term of the list
- The list is split into two smaller sublists (of equal size or one has one fewer term)
- Continue by restricting the search to the appropriate sublist
- Search for 19 in the (sorted) list

  1 2 3 5 6 7 8 10 12 13 15 16 18 19 20 22

Binary search

- First split the list

  1 2 3 5 6 7 8 10 12 13 15 16 18 19 20 22
- Then compare 19 and the largest term in the first list, and determine to use the list
- Continue

  12 13 15 16 18 19 20 22

  18 19 20 22

  19 (down to one term)

Binary search

procedure *binary search*(*x:integer,* a_1, a_2, …, a_n: increasing integers)

i:=1 (left endpoint of search interval)

j:=1 (right end point of search interval)

while (*i*<*j*)

begin

m:=$\lfloor (i+j)/2 \rfloor$

if x>a_m **then** i:=m+1

else j:=m

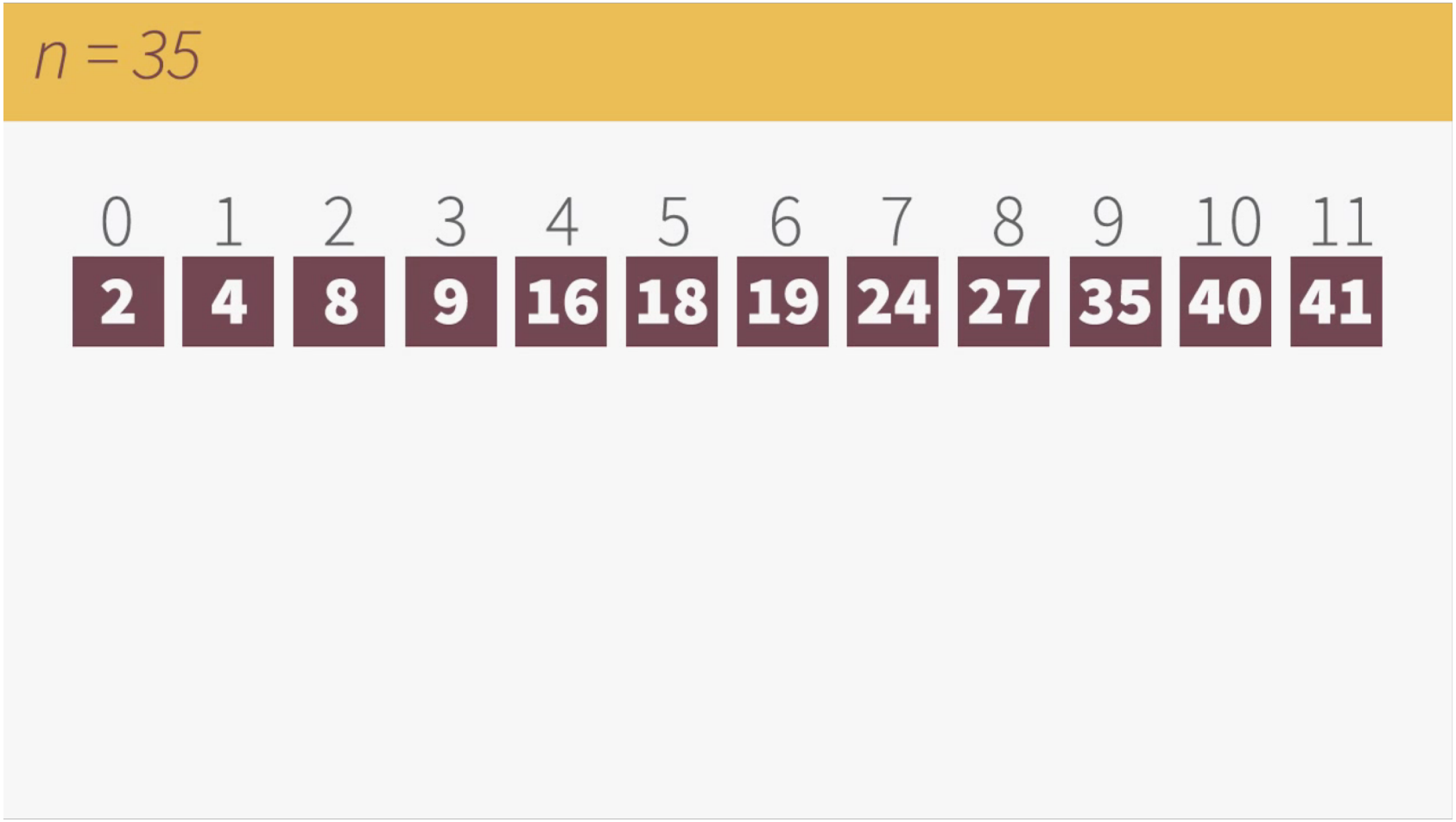
end

if *x*=a_i **then** *location*:=*i*

else *location:=0*

{*location* is the index of the term equal to x, or is 0 if x is not found}

Binary Search Animation



Sorting

- To *sort* the elements of a list is to put them in increasing order (numerical order, alphabetic, and so on).
- Sorting is very important problem both from practical and theoretical points of view.
- There is no “ideal” sorting algorithm.
- See, https://www.toptal.com/developers/sorting-algorithms

Bubble Sort

- *Bubble sort* makes multiple passes through a list. Every pair of elements that are found to be out of order are interchanged.

procedure *bubblesort*(a_1,…,a_n: real numbers with $n \geq 2$)

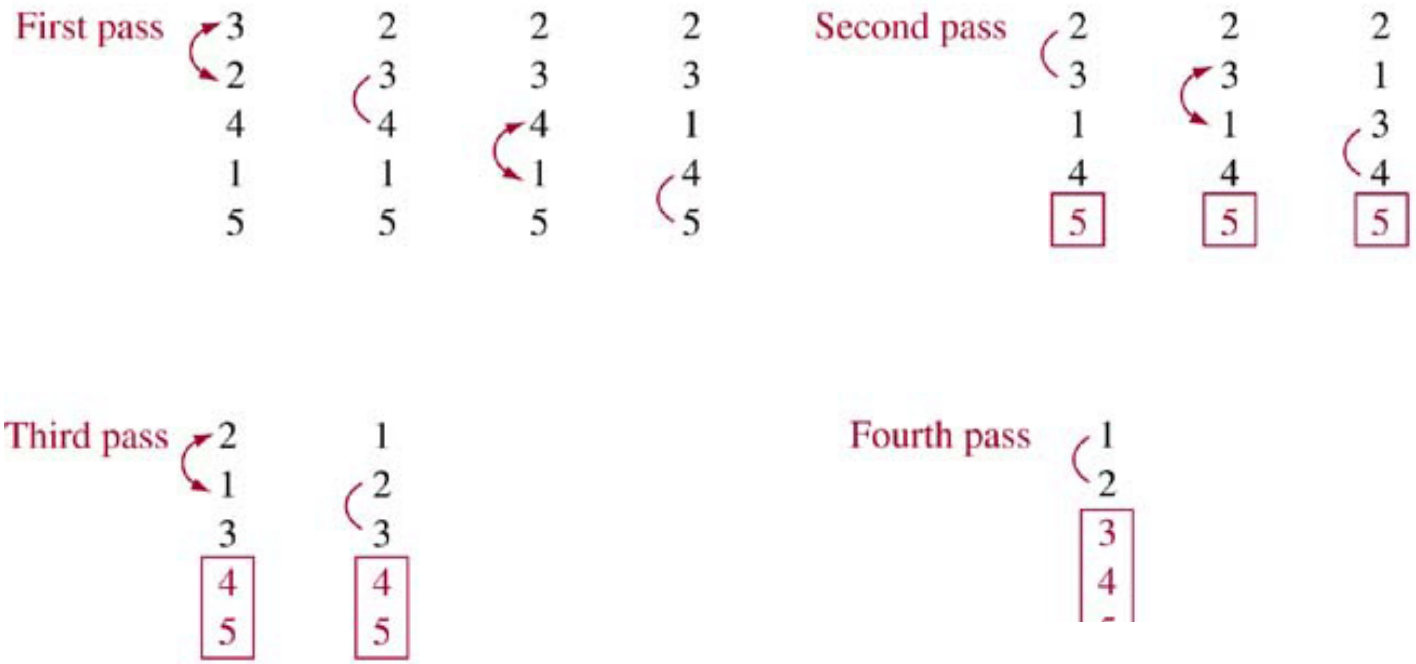
  for $i := 1$ to $n-1$

    for $j := 1$ to $n-i$

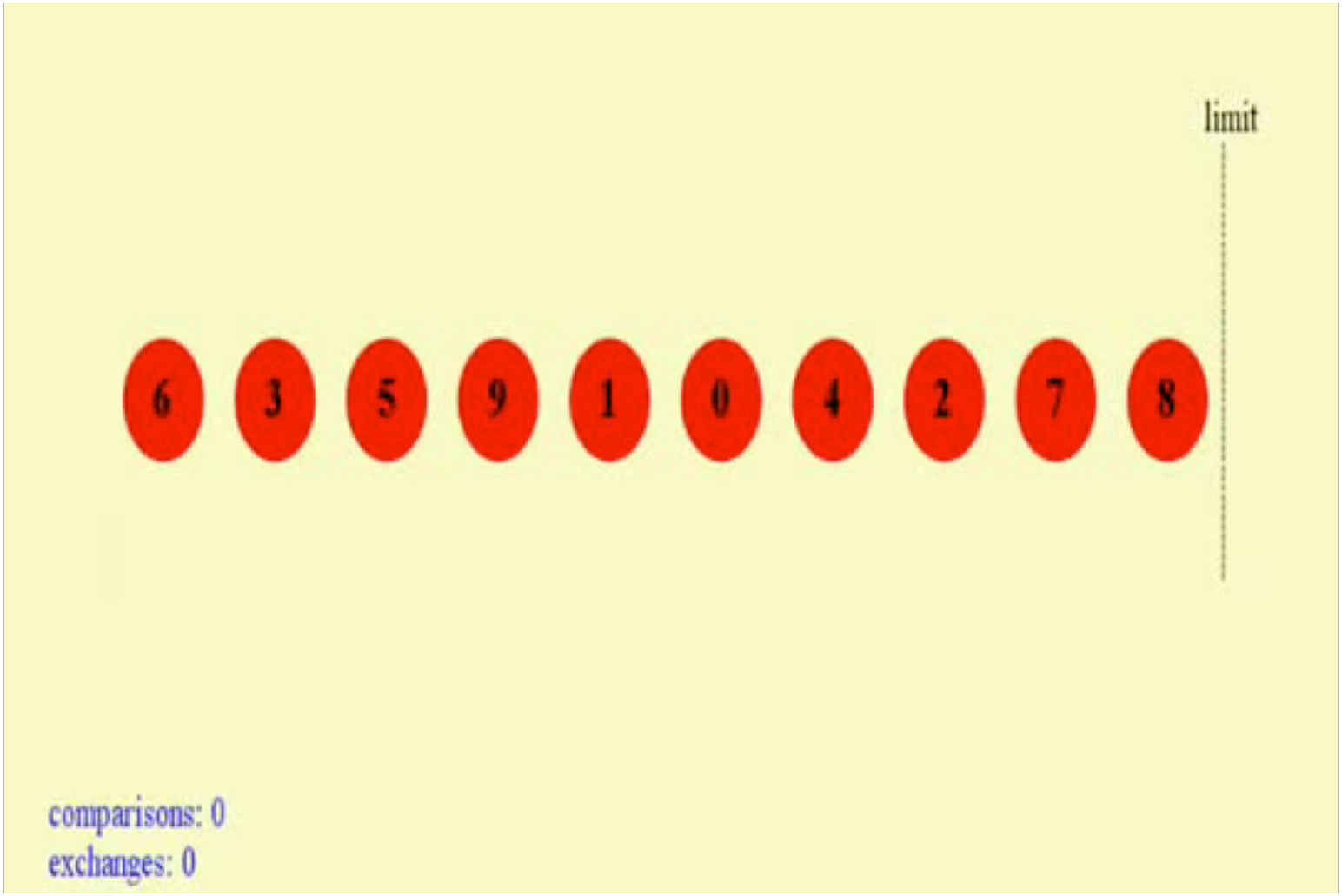
      if $a_j > a_{j+1}$ **then** interchange a_j and a_{j+1}

{a_1,…, a_n is now in increasing order}

Bubble Sort



Bubble Sort Animation



Insertion Sort

- *Insertion sort* begins with the 2^{nd} element. It compares the 2^{nd} element with the 1^{st} and puts it before the first if it is not larger.
- Next the 3^{rd} element is put into the correct position among the first 3 elements.
- In each subsequent pass, the $n+1^{st}$ element is put into its correct position among the first $n+1$ elements.
- Linear search is used to find the correct position.

Insertion Sort

procedure *insertion sort*
($a_1,\ldots,a_n$:
real numbers with $n \geq 2$)
for $j := 2$ to n
$i := 1$
while $a_j > a_i$
$i := i + 1$
$m := a_j$
for $k := 0$ to $j - i - 1$
$a_{j-k} := a_{j-k-1}$
$a_i := m$
{Now $a_1,\ldots,a_n$ is in increasing order}

Example

- Apply insertion sort to 3, 2, 4, 1, 5
- First compare 3 and 2 → *2, 3*, 4, 1, 5
- Next, insert 3rd item, 4>2, 4>3 → *2, 3, 4*, 1, 5
- Next, insert 4th item, 1<2 → *1, 2, 3, 4*, 5
- Next, insert 5th item, 5>1, 5>2, 5>3, 5>4→*1, 2, 3, 4, 5*

Insertion Sort Animation

Optimization Problems

- *Optimization problems* minimize or maximize some parameter over all possible inputs.
- Among the many optimization problems we will study are:
  - Finding a route between two cities with the smallest total mileage.
  - Determining how to encode messages using the fewest possible bits.
  - Finding the fiber links between network nodes using the least amount of fiber.

Greedy algorithm

- Many algorithms are designed to solve optimization problems
- Greedy algorithm:
  - Simple and naïve
  - Select the best choice at each step, instead of considering all sequences of steps
  - Once find a feasible solution
  - Either prove the solution is optimal or show a counterexample that the solution is non-optimal

Example

- Given n cents change with quarters, dimes, nickels and pennies, and use the least total number of coins
- Say, 67 cents
- Greedy algorithm
  - First select a quarter (leaving 42 cents)
  - Second select a quarter (leaving 17 cents)
  - Select a dime (leaving 7 cents)
  - Select a nickel (leaving 2cents)
  - Select a penny (leaving 1 cent)
  - Select a penny

Greedy change-making algorithm

procedure *change*(c_1, c_2, …, c_n: values of denominations of coins, where $c_1 > c_2 > \ldots > c_n$; n: positive integer)

for i:=1 **to** r

  while $n \geq c_i$ **then**

    add a coin with value c_i to the change

    n:=n- c_i

  end

Example

- Change of 30 cents
- If we use only quarters, dimes, and pennies (no nickels)
- Using greedy algorithm:
  - 6 coins: 1 quarter, 5 pennies
  - Could use only 3 coins (3 dimes)

Growth Function and Complexity

What Is the Best?

- We saw that there are several algorithms to solve certain problems. Some works better than other. What does “better” mean? How to evaluate the algorithm?
- Speed depends on the computer and on the input data. It should be something universal, behavioral to represent the quality of an algorithm.

Worst-case Analysis

- *Recall:*

```
procedure max(a_1, a_2, ...., a_n: integers)
    max := a_1
    for i := 2 to n
        if max < a_i
                    then max := a_i
    return max{max is the largest element}
```

- Let consider one line as one executable instruction, then $f(n) = 2 + 3(n\text{-}1) = 3n - 1$ is the number of instructions (steps) of this algorithm.

Worst-case Analysis

- In fact, the number of instructions to be executed depends on the data. For the data set A = { 4, 3, 2, 1 } we have one assignment and 3 tests. For the data set A = { 1, 2, 3, 4 } we have an assignment after each test.
- This is called “worst-case scenario” and the algorithms should be analyzed under such heavy-duty load.

Big Data

- Worst-case is a qualitative parameter of the input data.
- The other challenge is size of the input data.
- For the purposes of Computer Science it is always of interest how algorithms manage growing data.
- Let's think about function $f(n) = 3n - 1$ as a function that represents how the algorithm reacts on increasing of size of the input data without bounds.

Big Data

- It is easy to simplify the function to the form of $f(n) = 3n$ because impact of 1 is practically nothing in case of large values of *n.*
- What is the meaning of coefficient 3? It is a result of our assumption that each line of the code is one instruction. To clean the function from the implementation details we have to end up with $f(n) = n$.

Asymptotic Behavior

- Such function represents the **asymptotic behavior** of the algorithms.
- Any algorithm that doesn't have any loops will have $f(n) = 1$, since the number of instructions it needs is just a constant (unless it uses recursion).

Asymptotic Behavior

- Any program with a single loop which goes from 1 to n will have f(n) = n, since it will do a constant number of instructions before the loop, a constant number of instructions after the loop, and a constant number of instructions within the loop which all run n times.

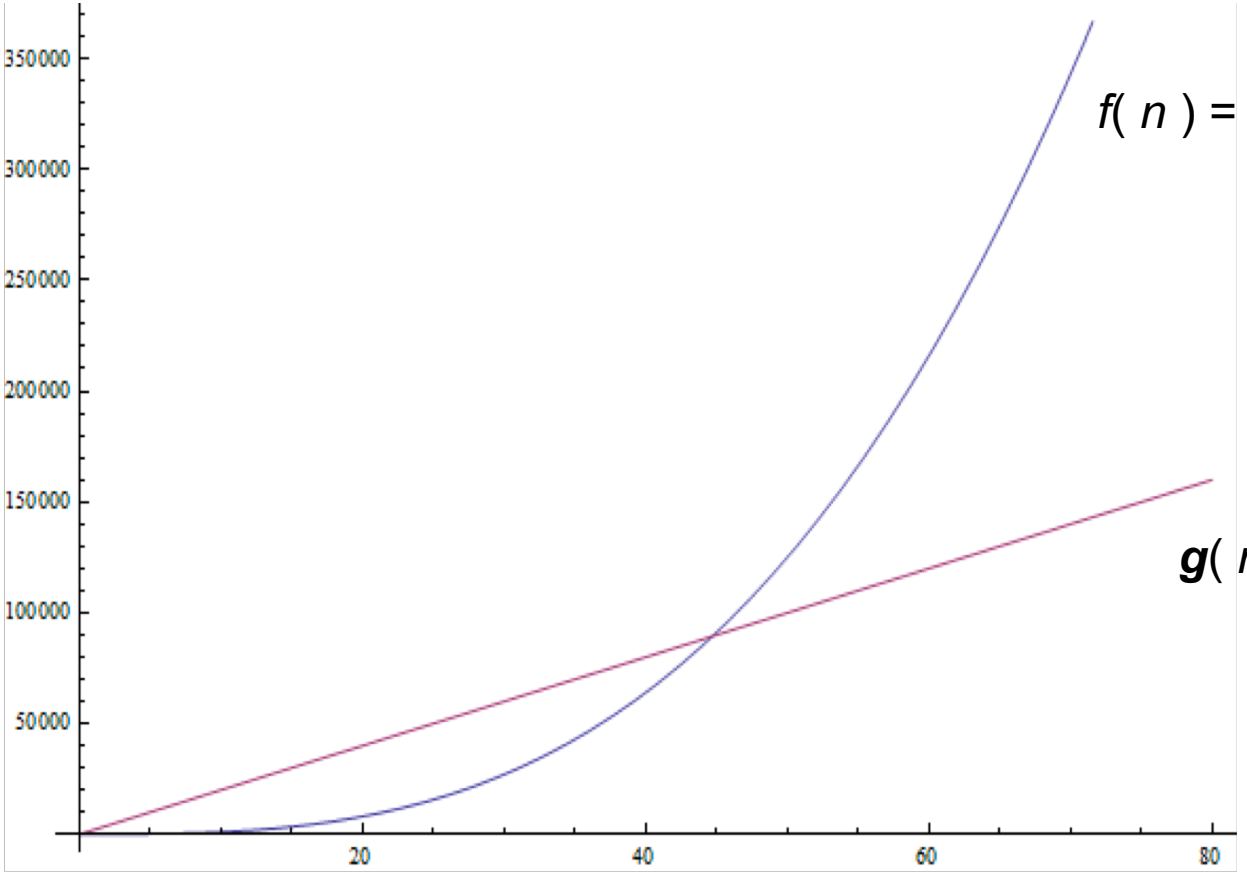
Asymptotic Behavior, cont'd

- Simple programs can be analyzed by counting the nested loops of the program:
  - A single loop over n items yields f(n) = n.
  - A loop within a loop yields f(n) = n^2.
  - A loop within a loop within a loop yields $f(n) = n^3$.
- Given a series of the loops that are sequential, the **slowest** of them determines the asymptotic behavior of the whole algorithm. Two nested loops followed by a single loop is asymptotically the same as the nested loops alone, because the nested loops dominate the simple loop.

Example

- **Problem:** Find the asymptotic behavior of the following function:
- $f(n) = n^3 + 1999n + 1337$
- **Solution:** $f(n) = n^3$
- Even though the factor in front of n is quite large, we can still find a large enough n so that n^3 is bigger than $1999n$. As we're interested in the behavior for very large values of n, we only keep n^3
- The n^3 function, drawn in blue, becomes larger than the $1999n$ function, drawn in purple, after $n = 45$. After that point it remains larger for ever.

Example


350000
300000
250000
200000
150000
100000
50000
20
40
60
80
f(n) = n3
g(n) = 1999n

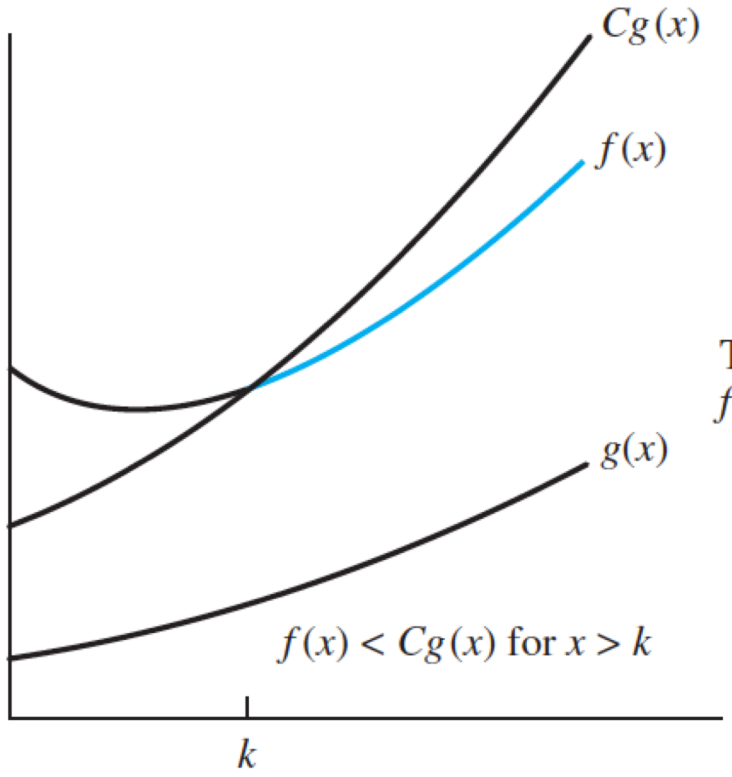
Big-*O* Notation

- Let *f* and *g* be functions from the set of integers or the set of real numbers to the set of real numbers. We say that *f*(*x*) is *O*(*g*(*x*)) if there are constants *C* and *k* such that

$$|f(x)| \leq C|g(x)|$$

  whenever $x > k$.
- This is read as "*f*(*x*) is big oh of *g*(*x*)" or "*g* asymptotically dominates *f*."
- The constants C and k are called **witnesses** to the relationship *f*(*x*) is *O*(*g*(*x*)). Only one pair of witnesses is needed.

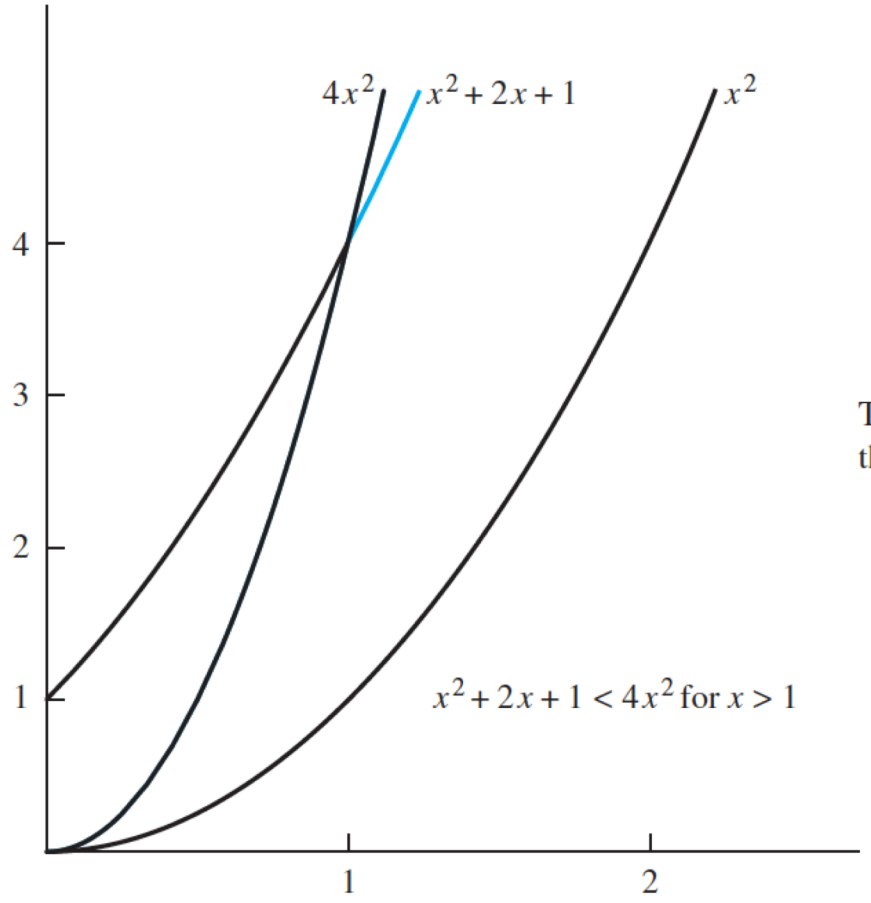
Big-O Notation



The part of the graph of $f(x)$ that satisfies $f(x) < Cg(x)$ is shown in color.

FIGURE 2 **The Function $f(x)$ is $O(g(x))$.**

Example



The part of the graph of $f(x) = x^2 + 2x + 1$ that satisfies $f(x) < 4x^2$ is shown in blue.

FIGURE 1 **The Function $x^2 + 2x + 1$ is $O(x^2)$.**

Important Notes about Big-*O* Notation

- If one pair of witnesses is found, then there are infinitely many pairs. We can always make the *k* or the *C* larger and still maintain the inequality

$$|f(x)| \leq C|g(x)|$$

- You may see " $f(x) = O(g(x))$" instead of " $f(x)$ is $O(g(x))$" but it is not an equation. It is ok to write $f(x) \in O(g(x))$, because $O(g(x))$ represents the set of functions that are $O(g(x))$.

- Usually, we will drop the absolute value sign since we will always deal with functions that take on positive values.

Practice

- **Problem**: Use big-*O* notation to estimate the sum of the first *n* positive integers.
- **Solution**:

$$1 + 2 + \cdots + n \leq n + n + \cdots n = n^2$$

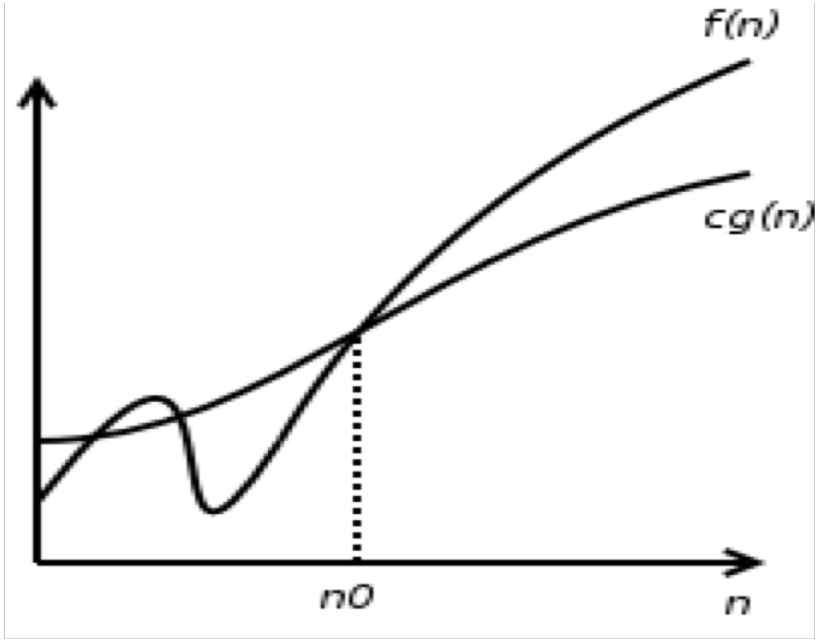
$$1 + 2 + \ldots + n \text{ is } O(n^2) \text{ taking } C = 1 \text{ and } k = 1.$$

Big-Omega Notation

- Let *f* and *g* be functions from the set of integers or the set of real numbers to the set of real numbers. We say that $f(x)$ is $\Omega(g(x))$
- if there are constants *C* and *k* such that $|f(x)| \geq C|g(x)|$
- when $x > k$.
- We say that "*f*(*x*) is big-Omega of *g*(*x*)."

Big-Omega Notation

- Big-*O* gives an upper bound on the growth of a function, while Big-Omega gives a lower bound. Big-Omega tells us that a function grows at least as fast as another.
- $f(x)$ is $\Omega(g(x))$ if and only if $g(x)$ is $O(f(x))$. This follows from the definitions.

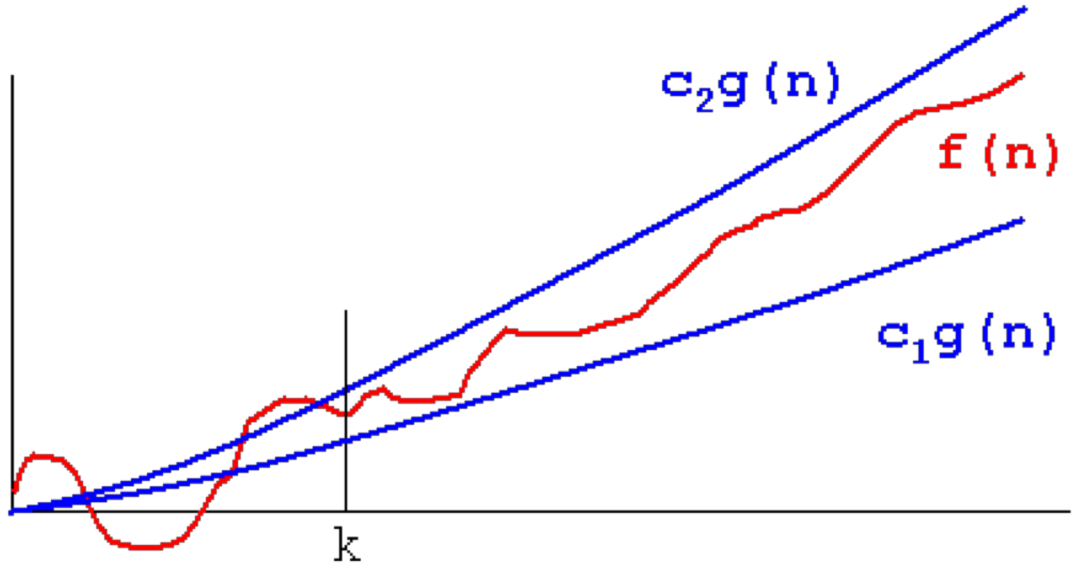


Big-Theta Notation

- Find the Asymptotic Behavior:
- $f(n) = n^6 + 3n$ $\quad n^6 + 3n \in \Theta(n^6)$
- $f(n) = 2^n + 12$ $\quad 2^n + 12 \in \Theta(2^n)$
- $f(n) = 3^n + 2^n$ $\quad 3^n + 2^n \in \Theta(3^n)$
- $f(n) = n^n + n$ $\quad n^n + n \in \Theta(n^n)$
- Once we found asymptotic behavior *g* for the function *f* we denote this by
  - $f(n)$ is $\Theta(g(n))$ ”f is theta of g".
- There is an alternative notation: $2n \in \Theta(n)$ pronounced as "two n is theta of n” and means that if the number of instructions of the algorithm is $2n$, then the asymptotic behavior of the algorithm is described by n.

Big-Theta Notation

- Let *f* and *g* be functions from the set of integers or the set of real numbers to the set of real numbers. The function $f(x)$ is $\Theta(g(x))$ if $f(x)$ is $O(g(x))$ and $f(x)$ is $\Omega(g(x))$



Big-Theta Notation

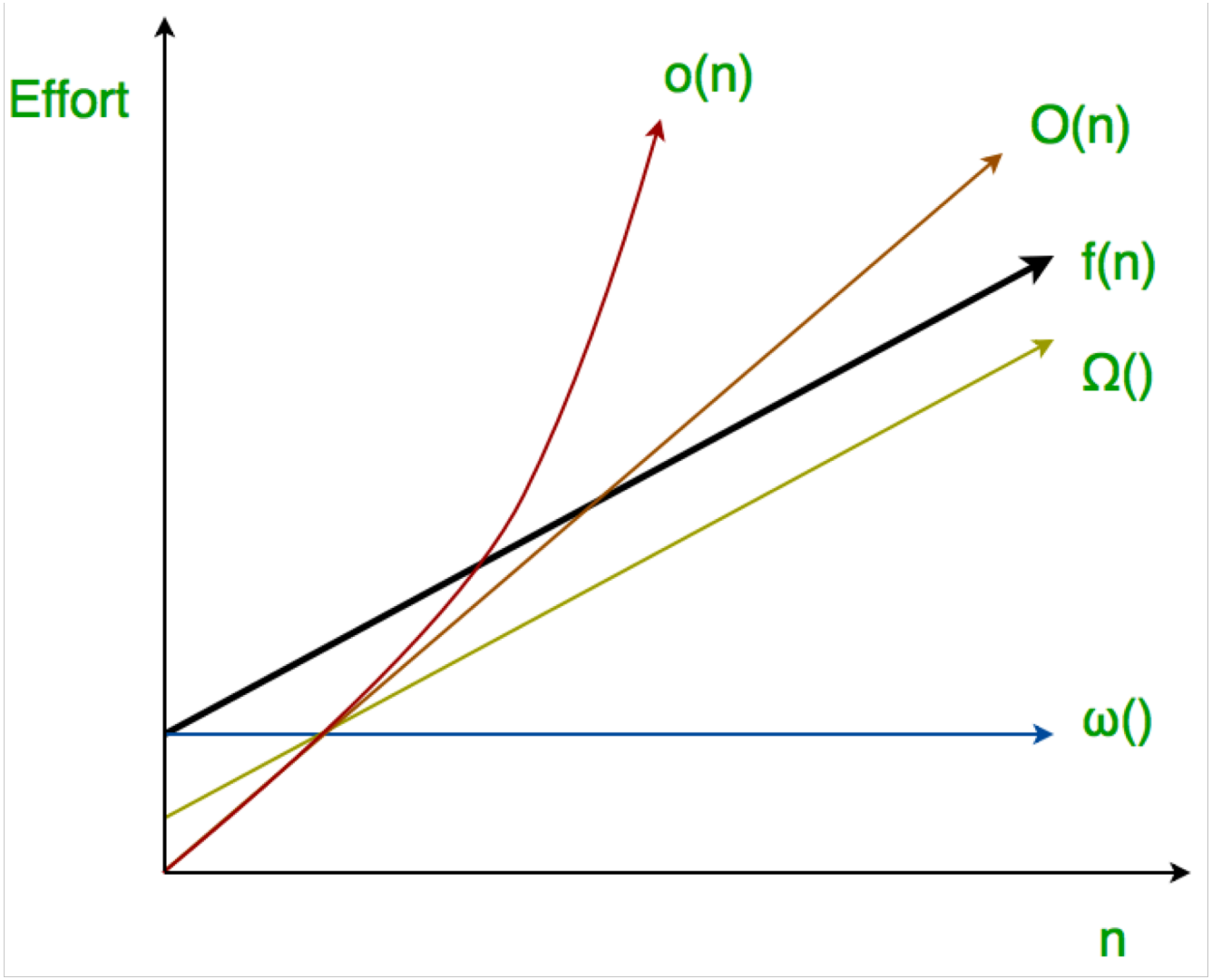
- We say that "f is big-Theta of $g(x)$" and also that "$f(x)$ is of *order* $g(x)$" and also that "$f(x)$ and $g(x)$ are of the *same order*."

- $f(x)$ is $\Theta(g(x))$ if and only if there exists constants C_1 , C_2 and k such that $C_1 g(x) < f(x) < C_2 g(x)$ if $x > k$. This follows from the definitions of big-*O* and big-Omega.

Little - Oh and Little - Omega

- If $f(x)$ is $O(g(x))$, but not $\Theta(g(x))$, then $f(x)$ is said to be $o(g(x))$,
  and it is read as "f(x) is little-oh of g(x)."
- For example, x is $o(x^2)$, x^2 is $o(2^x)$, 2^x is $o(x!)$.
- Similarly for little-omega ϖ .

Big-O, Big-Omega and Big-Theta



Time Complexity

- Big Theta defines time complexity. An algorithm with Θ($g(n)$) is of complexity $g(n)$.
- There are special names for algorithms depending of the complexity
  - $\Theta(1)$ is a constant-time algorithm,
  - $\Theta(n)$ is a linear algorithm,
  - $\Theta(n^2)$ is a quadratic algorithm,
  - $\Theta(\log(n))$ is a logarithmic algorithm.
- Programs with a bigger Θ run slower than programs with a smaller Θ.

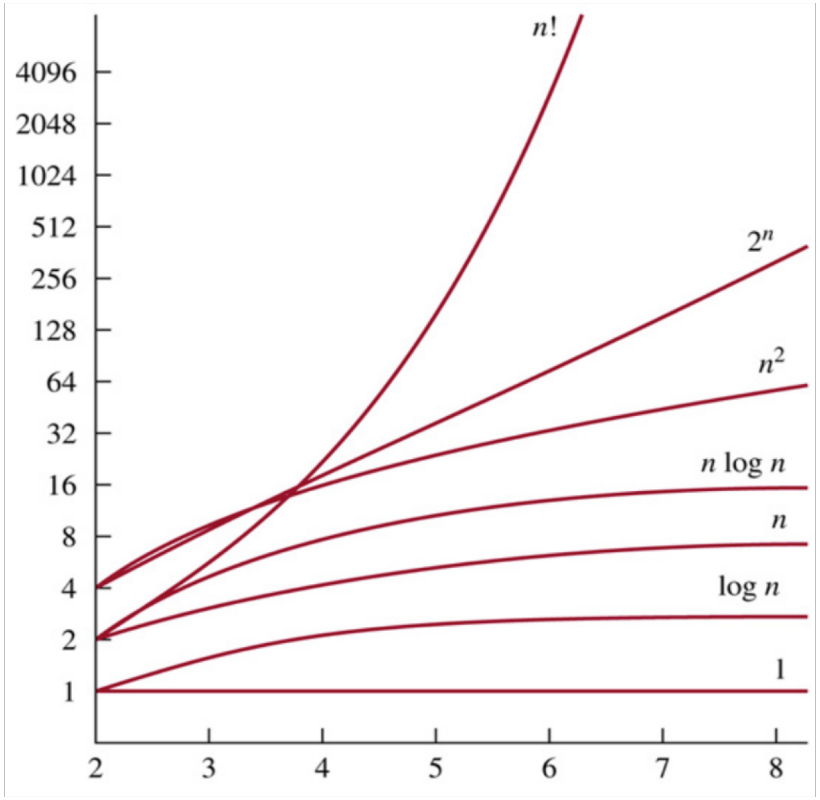
Comparison of Complexities

- Suppose we have three algorithms with complexities given by
  - the linear function n, drawn in green at the top, grows much faster than
  - the $\sqrt{n}$ function, drawn in red in the middle, which, in turn, grows much faster than
  - the $\log(n)$ function drawn in blue at the bottom of the plot.
- The conclusion is "execution time of a linear algorithm grows much faster than others with increasing of a size of data."

Comparison of Complexities

- Important complexity classes:

$$O(1) \subseteq O(\log n) \subseteq O(n) \subseteq O(n \log n) \subseteq O(n^2) \subseteq O(c^n) \subseteq O(n!)$$



A Display of the Growth of Functions Commonly Used in Big-O Estimates

Complexity of the Binary Search

- Let's assume, for simplicity, that the array is always cut in exactly a half.
- The number of elements to search in each iteration:
  - 0^{th} iteration: n
  - 1^{st} iteration: n / 2
  - 2^{nd} iteration: n / 4
  - 3^{rd} iteration: n / 8

  ...
  - i^{th} iteration: n / 2^i

  ...
  - last iteration: 1

Complexity of the Binary Search, cont'd

- The worst-case scenario is the the value we're looking for is the last or does not exist.
- The number of iteration for the worst-case is a solution of the equation:

  $1 = n / 2^i$

  $2^i = \mathrm{n}$

  $i = \log_2(n)$
- Therefore, the complexity of binary search is $\Theta(\log_2(n))$.
- This last result allows us to compare binary search with linear search. Clearly, as log(n) is much smaller than n, it is reasonable to conclude that binary search is a much faster method to search within an array then linear search, so it may be advisable to keep our arrays sorted if we want to do many searches within them.

Estimates of the Complexity

- In some many cases it is not straightforward to find the complexity of algorithm.
- **Example**: suppose that number of iteration of the algorithm is given by the factorial function

$$f(n) = n! = 1 \times 2 \times \cdots \times n \, .$$

- **Solution**: we may estimate the upper bound of the complexity of such algorithm if there is a function that grows at least not slower than *f*.

$$n! = 1 \times 2 \times \cdots \times n \leq n \times n \times \cdots \times n = n^n$$

or

$$n! \text{ is } O(n^n) \text{ taking } C = 1 \text{ and } k = 1.$$

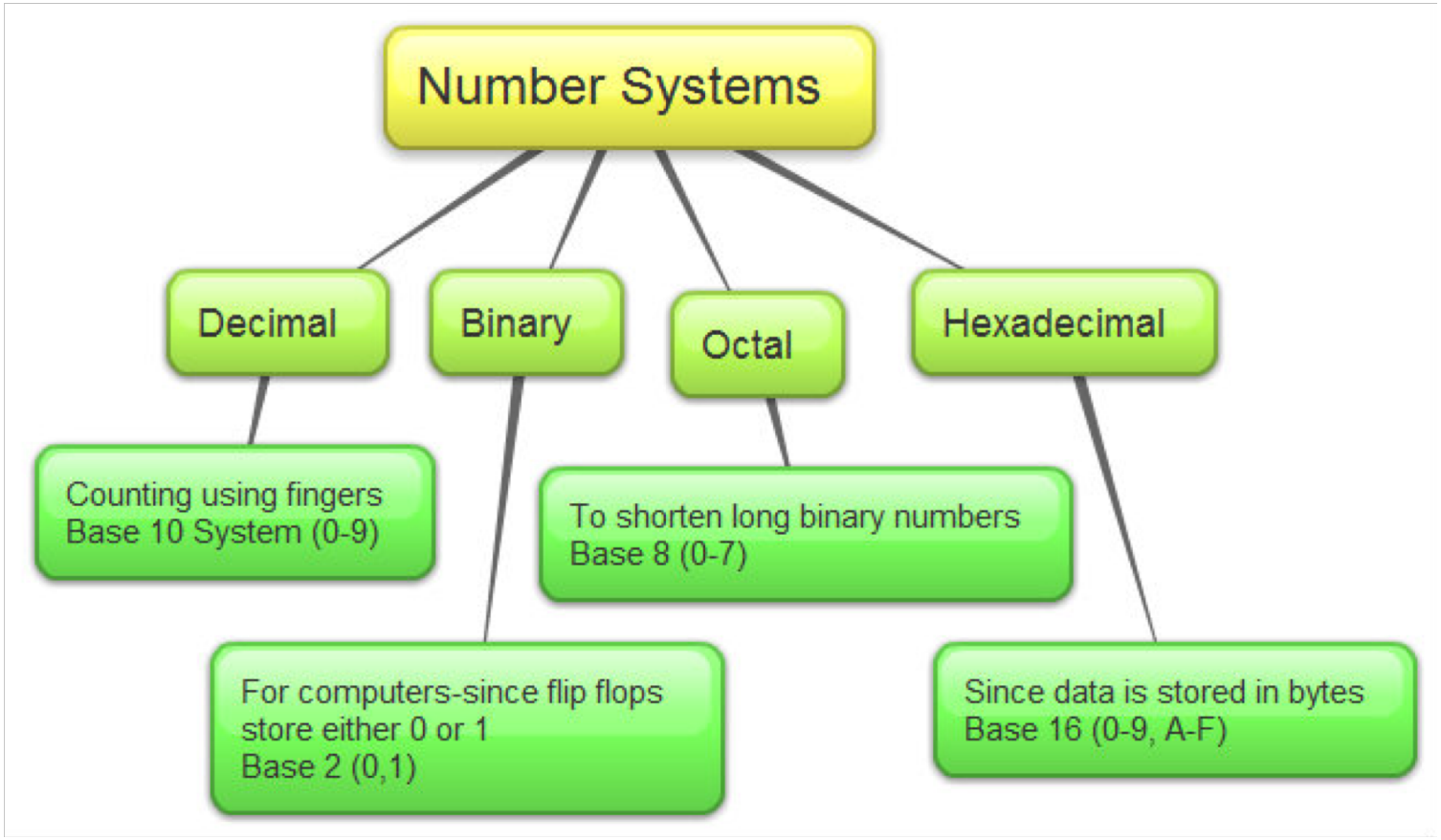
Integers and Division

Number theory

- **Number theory** is a branch of mathematics that explores integers and their properties.

- Integers:
  - – Z integers {…, -2,-1, 0, 1, 2, …}
  - – Z+ positive integers {1, 2, …}

Representations of integers



Representations of integers

Decimal number	Binary representation	Octal representation	Hexadecimal representation
0	0	0	0
1	1	1	1
2	10	2	2
3	11	3	3
4	100	4	4
5	101	5	5
6	110	6	6
7	111	7	7
8	1000	10	8
9	1001	11	9
10	1010	12	A
11	1011	13	B
12	1100	14	C
13	1101	15	D
14	1110	16	E
15	1111	17	F
16	10000	20	10

Applications

- Number theory has many applications within computer science, including:
  - – Indexing - Storage and organization of data
  - – Encryption
  - – Error correcting codes
  - – Random numbers generators

- Key ideas in number theory include **divisibility** and the **primality** of integers.

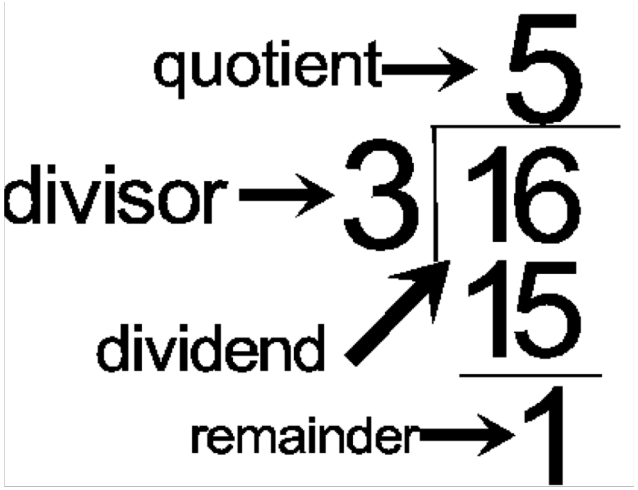
Why is division of integers so important?

Suppose that 35 friends are buying 200 tickets from you. How to do this and keep friendship?

Division Algorithm

- If a is an integer and d is a positive integer, then there are unique integers q and r, with $0 \leq r < d$, such that $a = dq + r$.

  - d is called the *divisor*.
  - a is called the *dividend*.
  - q is called the *quotient*.
  - r is called the *remainder*.



Examples

- Read Division Algorithm carefully and answer the following questions.

- **Question:** What are the quotient and the remainder when 200 is divided by 35?
- **Answer**: The quotient is 5 and the remainder is 25.

- **Question:** What are the quotient and the remainder when − 200 is divided by 35?
- **Answer**: The quotient is − 6 and the remainder is 10.

Division

- **Definition**: Assume 2 integers a and b, such that $a \neq 0$ (a is not equal 0). We say that **a divides b** if there is an integer c such that **$b = ac$**.
- If a divides b we say that **a is a factor of b** and that **b is multiple of a**.
- The fact that a divides b is denoted as **$a \mid b$**.
- If *a* does not divide *b*, we write ***$a \nmid b$***.

Examples

4 | 24 True or False ?

True

- 4 is a factor of 24
- 24 is a multiple of 4

3 | 7 True or False ?

False

Divisibility

Prove that if *a* is an integer other than 0, then

- 1 divides a.
- a divides 0.

Divisibility

- All integers divisible by d>0 can be enumerated as:

  .., -kd, …, -2d, -d, 0, d, 2d, …, kd, …

Question:

Let n and d be two positive integers. How many positive integers not exceeding n are divisible by d?

$$0 < kd \leq n$$

Divisibility

Question:

Let n and d be two positive integers. How many positive integers not exceeding n are divisible by d?

$$0 < kd \leq n$$

Answer:

Count the number of integers kd that are less than n. What is the number of integers k such that $0 < kd \leq n$?

$0 < kd \leq n$ --> $0 < k \leq n/d$ Therefore, there are $\lfloor n/d \rfloor$ positive integers not exceeding n that are divisible by d.

Properties of Divisibility

Let a, b, and c be integers, where $a \neq 0$.

(1) If $a \mid b$ and $a \mid c$, then $a \mid (b + c)$;

(2) If $a \mid b$, then $a \mid bc$ for all integers c;

(3) If $a \mid b$ and $b \mid c$, then $a \mid c$.

Properties of Divisibility

Let a, b, and c be integers, where $a \neq 0$.

(1) If $a \mid b$ and $a \mid c$, then $a \mid (b + c)$;

(2) If $a \mid b$, then $a \mid bc$ for all integers c;

(3) If $a \mid b$ and $b \mid c$, then $a \mid c$.

Proof of (1): if a | b and a | c, then a | (b +c)

- From the definition of divisibility we get:
- b=au and c=av where u,v are two integers. Then
- (b+c) = au +av = a(u+v)
- Thus a divides b+c.

Properties of Divisibility

Let a, b, and c be integers, where $a \neq 0$.

(1) If $a \mid b$ and $a \mid c$, then $a \mid (b + c)$;

(2) If $a \mid b$, then $a \mid bc$ for all integers c;

(3) If $a \mid b$ and $b \mid c$, then $a \mid c$.

Proof of (2): if a | b, then a | bc for all integers c

- If a | b, then there is some integer u such that b = au.
- Multiplying both sides by c gives us bc = auc, so by definition, a | bc.
- Thus a divides bc

Properties of Divisibility

Let a, b, and c be integers, where $a \neq 0$.

(1) If $a \mid b$ and $a \mid c$, then $a \mid (b + c)$;

(2) If $a \mid b$, then $a \mid bc$ for all integers c;

(3) If $a \mid b$ and $b \mid c$, then $a \mid c$.

Proof of (3): if a | b and b | c, then a | c

- If a | b, then there is some integer u such that b = au.
- If b | c, then there is some integer k such that c = kb = kau=aku, so by definition, a | c.
- Thus a divides c

Meaning of Integer Division

- Understanding of division starts from natural numbers: how to represent one set as a union of several other equal sets.
- Division of natural numbers with remainder is a representing of the set as a union of other sets with equal number of elements plus one set that does not have enough elements to be equal with others.
- **Example:** 100 flowers arranged in the bunches of 12 will result in 8 bouquets and 4 flowers.

Meaning of Integer Division

- Any natural number *b* could be divided by any natural number *a* with remainder *r* such as
  - $b = qa + r$
- and there are three possible cases:
  - $a \mid b \rightarrow r = 0.$
  - $a > b \rightarrow q = 0,\ r = b.$
  - $a \nmid b \rightarrow q \in \mathbf{Z}^+,\ r \in \mathbf{Z}^+.$

Meaning of Integer Division, cont'd

- Meaning of the Integer Division of positive integers is covered by Integer Division of natural numbers.
- But what about negative integers divided by positive integer?

- **Example:**
- Suppose, there is a loan of $1000 (negative number for accounting) that should be paid by 7 co-borrowers equally and rounded to $1.
- That is 7x$143 = $1001.
- $1 is the remainder.

Integer Division of Negative Numbers

- **Algorithm:**

1. Find absolute values (modulus) of dividend *a* and divisor *b*.
2. Divide moduli.
3. If remainder of step 2 is 0, the answer is the number opposite to the result of step 2.
4. If remainder of step 2 is not 0 then add 1 to the quotient of the result of step 2 and find the opposite to it. It is the quotient *q*.
5. The remainder is $r = a - b \cdot q$.

Definitions of Functions **div** and **mod**

- There are special notation to define the quotient and the remainder of Integer Division of *a* by *d*:
  - $q = a \textbf{ div } d$
  - $r = a \textbf{ mod } d$

Examples

A. Find −17 div 5 and − 17 mod 5.

1. $|-17| = 17, |5| = 5$.
2. $17 \text{ div } 5 = 3, 17 \bmod 5 = 2$.
3. $-(3 + 1) = -4$. **$-17 \text{ div } 5 = -4$.**
4. $-17 - 5 \cdot (-4) = -17 - (-20) = -17 + 20 = 3$. **$-17 \bmod 5 = 3$.**

B. Find −1404 div 26 and −1404 mod 26.

Answer: $-1404 \text{ div } 26 = -54, -1404 \bmod 26 = 0$.

Modular Arithmetic

Division Algorithm – Theorem of Existence

- *Recall:*
- According to the Division Algorithm
  - $32 \bmod 7 = 4$ because $32 = 7 \cdot 4 + 4$.
  - $-10 \bmod 7 = 4$ because $-10 = 7 \cdot (-2) + 4$.
- What is common between 32 and -10?

  The remainder.

Modulus

- The concept of positive remainder allows mapping of any set of integers to its subset defined by any positive integer *m* (modulus)
  - f: $S \subseteq \mathbf{Z} \rightarrow \mathbf{Z}_m = \{0, \ldots, m\text{-}1\} \subseteq S$.

Modular Arithmetic

- Let $(a, b) \in Z^2$, $m \in Z^+$ then a is a congruent to b modulo m if m divides a – b.
  Notation: $a \equiv b \pmod{m}$.

- Let a and b be integers, and let m be a positive integer. Then $a \equiv b \pmod{m}$ if and only if $a \bmod m = b \bmod m$.

- The equivalent definitions of congruency:

Let m be a positive integer. The integers a and b are congruent modulo m if and only if

$$\exists k \in Z;\ a = b + km$$

(mod *m*) and mod *m*

- *a* mod *m* = *b* denotes function that results in *b*.
- *a* ≡ *b* (mod *m*) denotes statement ("equation") that could be true or false.

Example

- $17 \equiv 5 \pmod 6$?

  $6 | (17 - 5) = 12 \Rightarrow 17 \equiv 5 \pmod 6$

- $24 \equiv 14 \pmod 6$?

  6 does not divide 10

  $\Rightarrow$ 24 is not congruent to 14 (mod 6)

Algebraic Operations and Congruencies

- If $a \equiv b \pmod{m}$ holds then $c \cdot a \equiv c \cdot b \pmod{m}$, where c is any integer, holds.
- If $a \equiv b \pmod{m}$ holds then $c + a \equiv c + b \pmod{m}$, where c is any integer, holds.
- Dividing a congruence by an integer does not always produce a valid congruence.
- **Example**:
- The congruence $14 \equiv 8 \pmod{6}$ holds.
- But dividing both sides by 2 does not produce a valid congruence since $14/2 = 7$ and $8/2 = 4$, but $7 \not\equiv 4 \pmod{6}$.

Example

- $17 \equiv 5 \pmod 6$

  $6 \mid (17 - 5) = 12 \Rightarrow 17 \equiv 5 \pmod 6$

- $8 \equiv 14 \pmod 6$

  $6 \mid (8-14) = -6 \Rightarrow 8 \equiv 14 \pmod 6$

$17 + 8 \equiv 5 + 14 \pmod 6$? $\Rightarrow$ $25 \equiv 19 \pmod 6$?

$17 \times 8 \equiv 5 \times 14 \pmod 6$? $\Rightarrow$ $136 \equiv 70 \pmod 6$?

Congruencies of Sums and Products

Let m be a positive integer. If $a \equiv b \pmod{m}$ and $a \equiv b \pmod{m}$, then

$a + c \equiv b + d \pmod{m}$ and $ac \equiv bd \pmod{m}$

Example:

- Because $7 \equiv 2 \pmod{5}$ and $11 \equiv 1 \pmod{5}$,

  $18 = 7 + 11 \equiv 2 + 1 = 3 \pmod{5}$

- $77 = 7 \cdot 11 \equiv 2 \cdot 1 = 2 \pmod{5}$

Properties of **mod** Function

- Let m be a positive integer and let a and b be integers.
- Then
- $(a + b)$ (**mod** m) = ((a **mod** m) + (b **mod** m)) **mod** m

and

- ab **mod** m = ((a **mod** m) (b **mod** m)) **mod** m.

Practice

- It is often asked to find remainder of the positive powers of 10 over an integer number:

  $10 \bmod 3$, $10^2 \bmod 3$, $10^3 \bmod 3$, etc.

- or

  $10 \bmod 7$, $10^2 \bmod 7$, $10^3 \bmod 7$, etc.

Practice

- Use of the properties of the mod function reduce computing:
  - $10^n \bmod m = (10 \bmod m)^n \bmod m$, where $n \in \mathbf{Z}^+$.

- $10 \bmod 3 = 1 \rightarrow 10^n \bmod 3 = (10 \bmod 3)^n \bmod 3 = 1$ (Division Algorithm: $1 = 0 \times 3 + 1$)
- $10 \bmod 7 = 3 \rightarrow 10^n \bmod 7 = (10 \bmod 7)^n \bmod 7 = 3^n \bmod 7$

Computations

- Suppose we need to compute a^n mod m. It could be solved by sequences of multiplications and divisions. There is the faster approach.
- For example, we need to compute a^8 mod n. Instead of 7 multiplications and one reduction mod of a big number perform 3 multiplications and 3 mod operations of a smaller numbers:
  - a^8 mod m = $((a^2 \bmod m)^2 \bmod m)^2 \bmod m$.
- The same,
  - a^{16} mod m = $(((a^2 \bmod m)^2 \bmod m)^2 \bmod m)^2 \bmod m$.

Arithmetic Modulo *m*

Definitions: Let $\mathbf{Z}_m$ be the set of nonnegative integers less than m: $\{0,1, \ldots, m-1\}$

- The operation $+_m$ is defined as $a +_m b = (a + b)$ **mod** m. This is *addition modulo m*.
- The operation $\cdot_m$ is defined as $a \cdot_m b = (a \cdot b)$ **mod** m. This is *multiplication modulo m*.
- Using these operations is said to be doing *arithmetic modulo m*.

Example

- **Example**: Find $7 +_{11} 9$ and $7 \cdot_{11} 9$.

  Solution: Using the definitions above:

  - $7 +_{11} 9 = (7 + 9) \mod 11 = 16 \mod 11 = 5$
  - $7 \cdot_{11} 9 = (7 \cdot 9) \mod 11 = 63 \mod 11 = 8$

Properties of Arithmetic Modulo *m*

- The operations $+_m$ and $\cdot_m$ satisfy many of the same properties as ordinary addition and multiplication if operands belong to $\mathbf{Z}_m$.

  - *Closure*: If a and b belong to $\mathbf{Z}_m$, then $a +_m b$ and $a \cdot_m b$ belong to $\mathbf{Z}_m$.

  - *Associativity*: If a, b, and c belong to $\mathbf{Z}_m$, then $(a +_m b) +_m c = a +_m (b +_m c)$ and $(a \cdot_m b) \cdot_m c = a \cdot_m (b \cdot_m c)$.

  - *Commutativity*: If a and b belong to $\mathbf{Z}_m$, then $a +_m b = b +_m a$ and $a \cdot_m b = b \cdot_m a$.

  - *Identity elements*: The elements 0 and 1 are identity elements for addition and multiplication modulo m, respectively.
    - If a belongs to $\mathbf{Z}_m$, then $a +_m 0 = a$ and $a \cdot_m 1 = a$.

Properties of Arithmetic Modulo *m*, cont'd

- *Additive inverses*: If $a \neq 0$ belongs to $\mathbf{Z}_m$, then $m - a$ is the additive inverse of a modulo m and 0 is its own additive inverse.
  - $a +_m (m - a) = 0$ and $0 +_m 0 = 0$

- *Distributivity*: If a, b, and c belong to $\mathbf{Z}_m$, then
  - $a \cdot_m (b +_m c) = (a \cdot_m b) +_m (a \cdot_m c)$ and
    $(a +_m b) \cdot_m c = (a \cdot_m c) +_m (b \cdot_m c)$.

- Multiplicative inverses have not been included since they do not always exist. For example, there is no multiplicative inverse of 2 modulo 6

Practice

- **Problem:** Prove that A mod 3 = D mod 3, where A is a positive integer number, D is a sum of decimal digits of A.

- For example, 6371 mod 3 = 17 mod 3.

- **Solution:** Weighted Positional Notation $A = a_n 10^n + \ldots + a_1 10^1 + a_0 10^0$ (e.g. $6371 = 6 \times 10^3 + 3 \times 10^2 + 7 \times 10^1 + 1 \times 10^0$).
- $A \bmod 3 = (a_n \times 10^n + \ldots + a_1 \times 10^1 + a_0 \times 10^0) \bmod 3$

  $= ((a_n \times 10^n) \bmod 3 + \ldots + (a_1 \times 10^1) \bmod 3 + (a_0 \times 10^0) \bmod 3) \bmod 3$

Each term $(a_i \times 10^i) \bmod 3 = ((a_i \bmod 3) \times (10^i \bmod 3)) \bmod 3 = ((a_i \bmod 3) \times 1) \bmod 3$ (see Practice example above), i.e.

- $A \bmod 3 = ((a_n \bmod 3) \bmod 3 + \ldots + (a_1 \bmod 3) \bmod 3 + (a_0 \bmod 3) \bmod 3) \bmod 3 =$

  $= (a_n \bmod 3 + \ldots + a_1 \bmod 3 + a_0 \bmod 3) \bmod 3$

  $= (a_n + \ldots + a_1 + a_0) \bmod 3$

Results of Modular Arithmetic

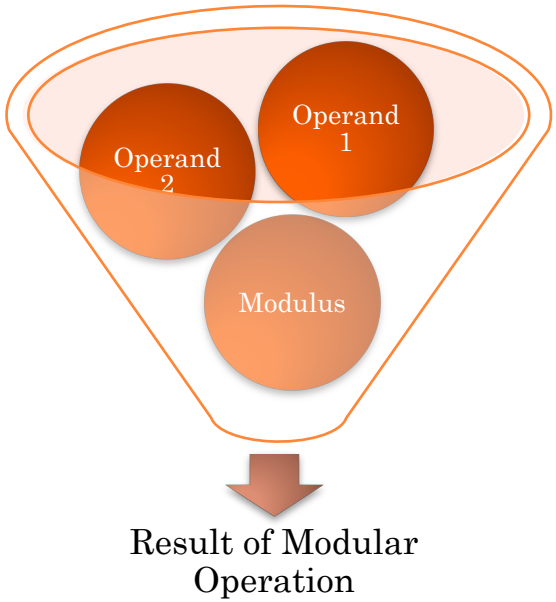
Module arithmetic replaces operations on integer numbers with operations on remainders of the division of the numbers by specified positive integer *m*.

It is obvious that different integers may result in the same numbers:

$$(16 \text{x} 2) \bmod 7 = 4$$

$$(20 - 30) \bmod 7 = 4$$

How to evaluate such results?



Applications of Congruences: Hash Functions

Assignment of memory location to a student record

$$h(k) = k \textbf{ mod } m$$

Key: social security #

# of available memory location

Example: h (064212848) = 064212848 mod 111 = 14 when m = 111

Applications of Congruences: Cryptology

a) Encryption:

- Making messages secrets by shifting each letter three letters forward in the alphabet

  $B \rightarrow E$ $\quad\quad$ $X \rightarrow A$

- Mathematical expression:

  $f(p) = (p + 3) \bmod 26 \quad 0 \leq p \leq 25$

Applications of Congruences: Cryptology

- Example: What is the secret message produced from the message "Meet you in the park"

Solution:

1. Replace letters with numbers:
   meet = 12 4 4 19
   you = 24 14 20
   in = 8 1 3
   the = 19 7 4
   park = 15 0 17 10
2. Replace each of these numbers p by $f(p) = (p + 3) \bmod 26$
   meet = 15 7 7 22
   you = 1 17 23
   in = 11 16
   the = 22 10 7
   park = 18 3 20 13
3. Translate back into letters: "PHHW BRX LQ WKH SDUN"

Applications of Congruences: Cryptology

b) Decryption (Deciphering)

$f(p) = (p + k)$ **mod** 26 (shift cepher)
$\Rightarrow f^{-1}(p) = (p - k)$ **mod** 26

Caesar's method and shift cipher are very vulnerable and thus have low level of security (reason frequency of occurrence of letters in the message)
$\Rightarrow$ Replace letters with blocks of letters.

Next class

- Topic: Number Theory
- Pre-class reading: Chap 4