Discrete Mathematics
Lec 4: Algorithms

馬誠佑

Chapter Summary

Algorithms

- Example Algorithms
- Algorithmic Paradigms

Growth of Functions

- Big-*O* and other Notation

Complexity of Algorithms

Algorithms

Section 3.1

Section Summary[1]

Properties of Algorithms

Algorithms for Searching and Sorting

Greedy Algorithms

Halting Problem

Problems and Algorithms

In many domains there are key general problems that ask for output with specific properties when given valid input.

The first step is to precisely state the problem, using the appropriate structures to specify the input and the desired output.

We then solve the general problem by specifying the steps of a procedure that takes a valid input and produces the desired output. This procedure is called an *algorithm*.

Algorithms

Abu Ja'far Mohammed
Ibin Musa Al-Khowarizmi
(780-850)

Definition: An *algorithm* is a finite set of precise instructions for performing a computation or for solving a problem.

Example: Describe an algorithm for finding the maximum value in a finite sequence of integers.

Solution: Perform the following steps:

1. Set the temporary maximum equal to the first integer in the sequence.
2. Compare the next integer in the sequence to the temporary maximum.
   - If it is larger than the temporary maximum, set the temporary maximum equal to this integer.
3. Repeat the previous step if there are more integers. If not, stop.
4. When the algorithm terminates, the temporary maximum is the largest integer in the sequence.

Specifying Algorithms

Algorithms can be specified in different ways. Their steps can be described in English or in *pseudocode.*

Pseudocode is an intermediate step between an English language description of the steps and a coding of these steps using a programming language.

The form of pseudocode we use is specified in Appendix 3. It uses some of the structures found in popular languages such as C++ and Java.

Programmers can use the description of an algorithm in pseudocode to construct a program in a particular language.

Pseudocode helps us analyze the time required to solve a problem using an algorithm, independent of the actual programming language used to implement algorithm.

Properties of Algorithms

Input: An algorithm has input values from a specified set.

Output: From the input values, the algorithm produces the output values from a specified set. The output values are the solution.

Correctness: An algorithm should produce the correct output values for each set of input values.

Finiteness: An algorithm should produce the output after a finite number of steps for any input.

Effectiveness: It must be possible to perform each step of the algorithm correctly and in a finite amount of time.

Generality: The algorithm should work for all problems of the desired form.

Finding the Maximum Element in a Finite Sequence

The algorithm in pseudocode:

```
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}
```

Does this algorithm have all the properties listed on the previous slide?

Some Example Algorithm Problems

Three classes of problems will be studied in this section.

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

Searching Problems

Definition: 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.
- For example, a library might want to check to see if a patron is on a list of those with overdue books before allowing him/her to checkout another book.
- We will study two different searching algorithms: linear search and binary search.

Linear Search Algorithm

The linear search algorithm locates an item in a list by examining elements in the sequence one at a time, starting at the beginning.

- First compare x with a_1. If they are equal, return the position 1.
- If not, try a_2. If $x = a_2$, return the position 2.
- Keep going, and if no match is found when the entire list is scanned, return 0.

```
procedure linear search($x$:integer,
          $a_1, a_2, \ldots, a_n$: distinct integers)
$i$ := 1
while ($i \leq n$ and $x \neq a_i$)
    $i$ := $i + 1$
if $i \leq n$ then location := $i$
else location := 0
return location{location is the subscript of the term that
    equals $x$, or is 0 if $x$ is not found}
```

Binary Search 1

Assume the input is a list of items in increasing order.

The algorithm begins by comparing the element to be found with the middle element.

- If the middle element is lower, the search proceeds with the upper half of the list.
- If it is not lower, the search proceeds with the lower half of the list (through the middle position).

Repeat this process until we have a list of size 1.

- If the element we are looking for is equal to the element in the list, the position is returned.
- Otherwise, 0 is returned to indicate that the element was not found.

In Section 3.3, we show that the binary search algorithm is much more efficient than linear search.

Binary Search 2

Here is a description of the binary search algorithm in pseudocode.

```
procedure binary search($x$: integer, $a_1, a_2, \ldots, a_n$: increasing integers)
    $i := 1$ {$i$ is the left endpoint of interval}
    $j := n$ {$j$ is right endpoint of interval}
    while $i < j$
                        $m := \lfloor (i + j)/2 \rfloor$
              if $x > a_m$ then $i := m + 1$
              else $j := m$
    if $x = a_i$ then $location := i$
    else $location := 0$
    return $location${location is the subscript $i$ of the term $a_i$ equal to $x$,
                                            or $0$ if $x$ is not found}
```

Binary Search 3

Example: The steps taken by a binary search for 19 in the list:

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

1. The list has 16 elements, so the midpoint is 8. The value in the 8th position is 10. Since 19 > 10, further search is restricted to positions 9 through 16.

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

2. The midpoint of the list (positions 9 through 16) is now the 12th position with a value of 16. Since 19 > 16, further search is restricted to the 13th position and above.

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

3. The midpoint of the current list is now the 14th position with a value of 19. Since $19 \not> 19$, further search is restricted to the portion from the 13th through the 14th positions .

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

4. The midpoint of the current list is now the 13th position with a value of 18. Since 19> 18, search is restricted to the portion from the 14th position through the 14th.

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

5. Now the list has a single element and the loop ends. Since 19=19, the location 14 is returned.

Sorting

To *sort* the elements of a list is to put them in increasing order (numerical order, alphabetic, and so on).

Sorting is an important problem because:

- A nontrivial percentage of all computing resources are devoted to sorting different kinds of lists, especially applications involving large databases of information that need to be presented in a particular order (e.g., by customer, part number etc.).
- An amazing number of fundamentally different algorithms have been invented for sorting. Their relative advantages and disadvantages have been studied extensively.
- Sorting algorithms are useful to illustrate the basic notions of computer science.

A variety of sorting algorithms are studied in this book; binary, insertion, bubble, selection, merge, quick, and tournament.

In Section 3.3, we'll study the amount of time required to sort a list using the sorting algorithms covered in this section.

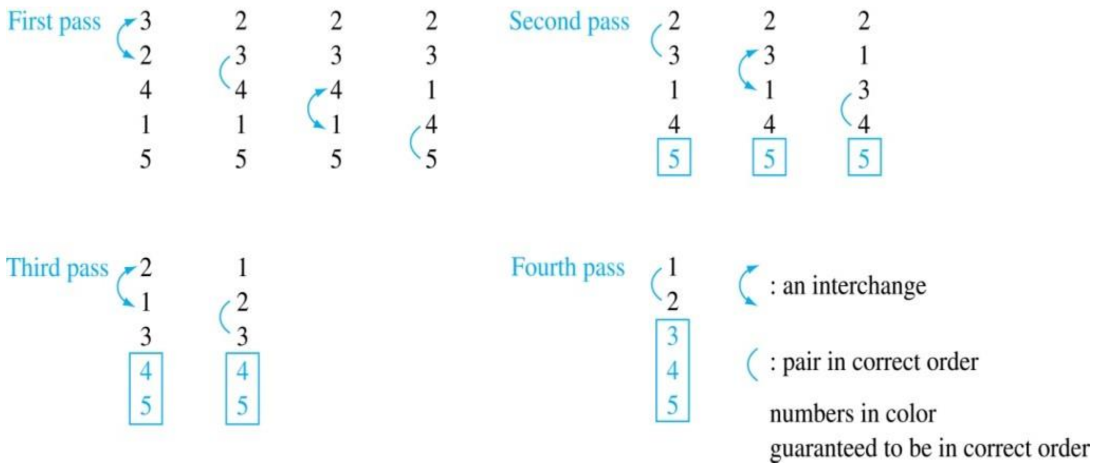
Bubble Sort $_1$

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

```
procedure bubblesort($a_1,\ldots,a_n$: 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 $a_{j+1}$
{$a_1,\ldots, a_n$ is now in increasing order}
```

Bubble Sort 2

Example: Show the steps of bubble sort with 3 2 4 1 5



At the first pass the largest element has been put into the correct position

At the end of the second pass, the 2nd largest element has been put into the correct position.

In each subsequent pass, an additional element is put in the correct position.

Jump to long description

Insertion Sort 1

Insertion sort begins with the 2nd element. It compares the 2nd element with the 1st and puts it before the first if it is not larger.

Next the 3rd element is put into the correct position among the first 3 elements.

In each subsequent pass, the n+1st element is put into its correct position among the first n+1 elements.

Linear search is used to find the correct position.

```
procedure insertion sort
($a_1,…,a_n$:
    real numbers with $n ≥ 2$)
    for $j := 2$ to $n$
            $i := 1$
            while $a_j < a_i$
            $i := i + 1$
            $m := a_j$
               for $k := 0$ to $j − k − 1$
                  $a_{j-k} := a_{j-k-1}$
                  $a_i := m$
{Now $a_1,…,a_n$ is in increasing
    order}
```

```
for i from 1 to length(arr) - 1:
      key = arr[i]
      j = i - 1
      while j >= 0 and arr[j] > key:
         arr[j + 1] = arr[j]
         j = j - 1
      arr[j + 1] = key
```

Insertion Sort $_2$

Example: Show all the steps of insertion sort with the input:
3 2 4 1 5

i. 2 3 4 1 5 (*first two positions are interchanged*)

ii. 2 3 4 1 5 (*third element remains in its position*)

iii. 1 2 3 4 5 (*fourth is placed at beginning*)

iv. 1 2 3 4 5 (*fifth element remains in its position*)

Greedy Algorithms

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.

Optimization problems can often be solved using a *greedy algorithm*, which makes the "best" choice at each step. Making the "best choice" at each step does not necessarily produce an optimal solution to the overall problem, but in many instances, it does.

After specifying what the "best choice" at each step is, we try to prove that this approach always produces an optimal solution, or find a counterexample to show that it does not.

The greedy approach to solving problems is an example of an algorithmic paradigm, which is a general approach for designing an algorithm. We return to algorithmic paradigms in Section 3.3.

Greedy Algorithms: Making Change

Example: Design a greedy algorithm for making change (in U.S. money) coins: quarters (25 cents), dimes (10 cents), nickels (5 cents), and pennies (1 cent) , using the least total number of coins.

Idea: At each step choose the coin with the largest possible value that does not exceed the amount of change left.

1. If *n* = 67 cents, first choose a quarter leaving 67−25 = 42 cents. Then choose another quarter leaving 42 −25 = 17 cents
2. Then choose 1 dime, leaving 17 − 10 = 7 cents.
3. Choose 1 nickel, leaving 7 – 5 = 2 cents.
4. Choose a penny, leaving one cent. Choose another penny leaving 0 cents.

Greedy Change-Making Algorithm 1

Solution: Greedy change-making algorithm for n cents. The algorithm works with any coin denominations $c_1, c_2, \ldots, c_r$.

```
procedure change($c_1, c_2, \ldots, c_r$: values of coins, where $c_1 > c_2 > \ldots > c_r$;
    $n$: a positive integer)
for $i$ := 1 to $r$
    $d_i := 0$ [$d_i$ counts the coins of denomination $c_i$]
    while $n \geq c_i$
            $d_i := d_i + 1$ [add a coin of denomination $c_i$]
            $n = n - c_i$
[$d_i$ counts the coins $c_i$]
```

For the example of U.S. currency, we may have quarters, dimes, nickels and pennies, with $c_1 = 25$, $c_2 = 10$, $c_3 = 5$, and $c_4 = 1$.

Proving Optimality for U.S. Coins[1]

Show that the change making algorithm for *U.S.* coins is optimal.

Lemma 1: If *n* is a positive integer, then *n* cents in change using quarters, dimes, nickels, and pennies, using the fewest coins possible has at most 2 dimes, 1 nickel, 4 pennies, and cannot have 2 dimes and a nickel. The total amount of change in dimes, nickels, and pennies must not exceed 24 cents.

Proof: By contradiction

- If we had 3 dimes, we could replace them with a quarter and a nickel.
- If we had 2 nickels, we could replace them with 1 dime.
- If we had 5 pennies, we could replace them with a nickel.
- If we had 2 dimes and 1 nickel, we could replace them with a quarter.
- The allowable combinations, have a maximum value of 24 cents; 2 dimes and 4 pennies.

Proving Optimality for U.S. Coins 2

Theorem: The greedy change-making algorithm for U.S. coins produces change using the fewest coins possible.

Proof: By contradiction.

1. Assume there is a positive integer *n* such that change can be made for *n* cents using quarters, dimes, nickels, and pennies, with a fewer total number of coins than given by the algorithm.

2. Then, $\acute{q} \leq q$ where $\acute{q}$ is the number of quarters used in this optimal way and q is the number of quarters in the greedy algorithm's solution. But this is not possible by Lemma 1, since the value of the coins other than quarters can not be greater than 24 cents.

3. Similarly, by Lemma 1, the two algorithms must have the same number of dimes, nickels, and quarters.

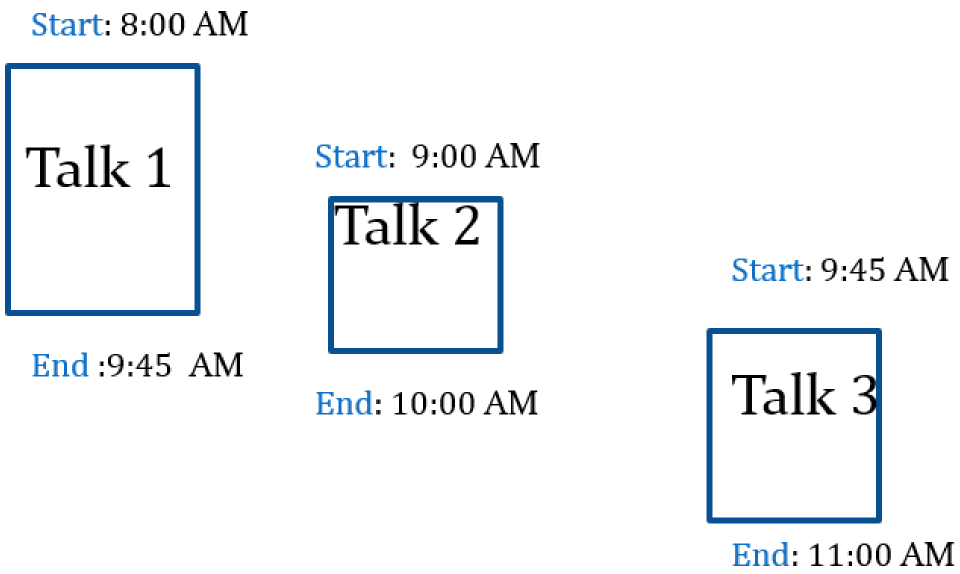
Greedy Scheduling [1]

Example: We have a group of proposed talks with start and end times. Construct a greedy algorithm to schedule as many as possible in a lecture hall, under the following assumptions:

- When a talk starts, it continues till the end.
- No two talks can occur at the same time.
- A talk can begin at the same time that another ends.
- Once we have selected some of the talks, we cannot add a talk which is incompatible with those already selected because it overlaps at least one of these previously selected talks.
- How should we make the "best choice" at each step of the algorithm? That is, which talk do we pick ?
  - The talk that starts earliest among those compatible with already chosen talks?
  - The talk that is shortest among those already compatible?
  - The talk that ends earliest among those compatible with already chosen talks?

Greedy Scheduling 2

Picking the shortest talk doesn’t work.



Start: 8:00 AM

Talk 1

End :9:45 AM

Start: 9:00 AM

Talk 2

End: 10:00 AM

Start: 9:45 AM

Talk 3

End: 11:00 AM

Can you find a counterexample here?

But picking the one that ends soonest does work.
The algorithm is specified on the next page.

Greedy Scheduling algorithm

Solution: At each step, choose the talks with the earliest ending time among the talks compatible with those selected.

```
procedure schedule($s_1 \leq s_2 \leq \ldots \leq s_n$ : start times, $e_1 \leq e_2 \leq \ldots \leq e_n$ : end
    times)
sort talks by finish time and reorder so that $e_1 \leq e_2 \leq \ldots \leq e_n$
$S := \emptyset$
for $j := 1$ to $n$
    if talk $j$ is compatible with $S$ then
            $S := S \cup \{\text{talk } j\}$
return S [ S is the set of talks scheduled]
```

Will be proven correct by induction in Chapter 5.

Halting Problem

Example: Can we develop a procedure that takes as input a computer program along with its input and determines whether the program will eventually halt with that input.

Solution: Proof by contradiction.

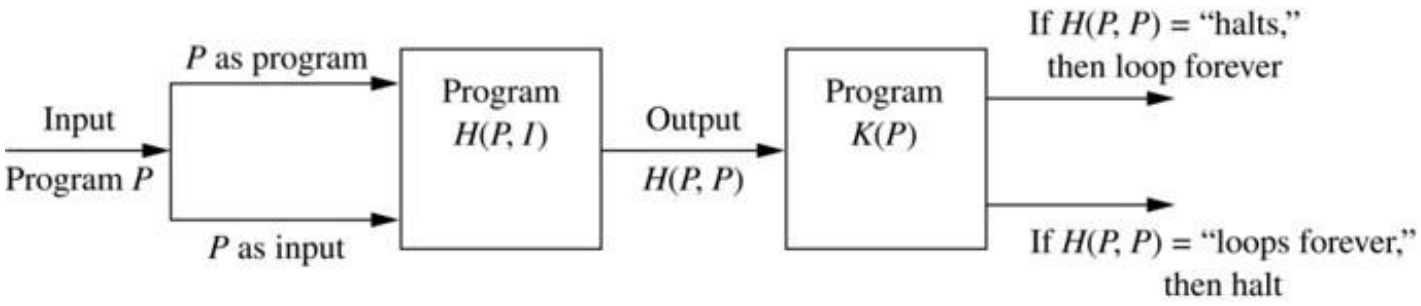
Assume that there is such a procedure and call it *H*(*P*,*I*). The procedure *H*(*P*,*I*) takes as input a program *P* and the input *I* to *P*.

- H outputs "halt" if it is the case that *P* will stop when run with input *I*.
- Otherwise, *H* outputs "loops forever."

Halting Problem[1]

Since a program is a string of characters, we can call *H*(*P*,*P*). Construct a procedure *K*(*P*), which works as follows.

- If *H*(*P*,*P*) outputs “loops forever” then *K*(*P*) halts.
- If *H*(*P*,*P*) outputs “halt” then *K*(*P*) goes into an infinite loop printing “ha” on each iteration.



Jump to long description

Halting Problem $_2$

Now we call *K* with *K* as input, i.e. *K*(*K*).

- If the output of *H*(*K*,*K*) is “loops forever” then *K*(*K*) halts. A Contradiction.
- If the output of *H*(*K*,*K*) is “halts” then *K*(*K*) loops forever. A Contradiction.

Therefore, there can not be a procedure that can decide whether or not an arbitrary program halts. The halting problem is unsolvable.

The Growth of Functions

Section 3.2

Section Summarys

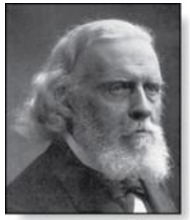
Donald E. Knuth
(Born 1938)

Big-O Notation

Big-O Estimates for Important Functions

Big-Omega and Big-Theta Notation

Edmund Landau
(1877-1938)



Paul Gustav Heinrich Bachmann
(1837-1920)

The Growth of Functions

In both computer science and in mathematics, there are many times when we care about how fast a function grows.

In computer science, we want to understand how quickly an algorithm can solve a problem as the size of the input grows.

- We can compare the efficiency of two different algorithms for solving the same problem.
- We can also determine whether it is practical to use a particular algorithm as the input grows.
- We'll study these questions in Section 3.3.

Two of the areas of mathematics where questions about the growth of functions are studied are:

- number theory (covered in Chapter 4)
- combinatorics (covered in Chapters 6 and 8)

Big-*O* Notation 1

Definition: 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

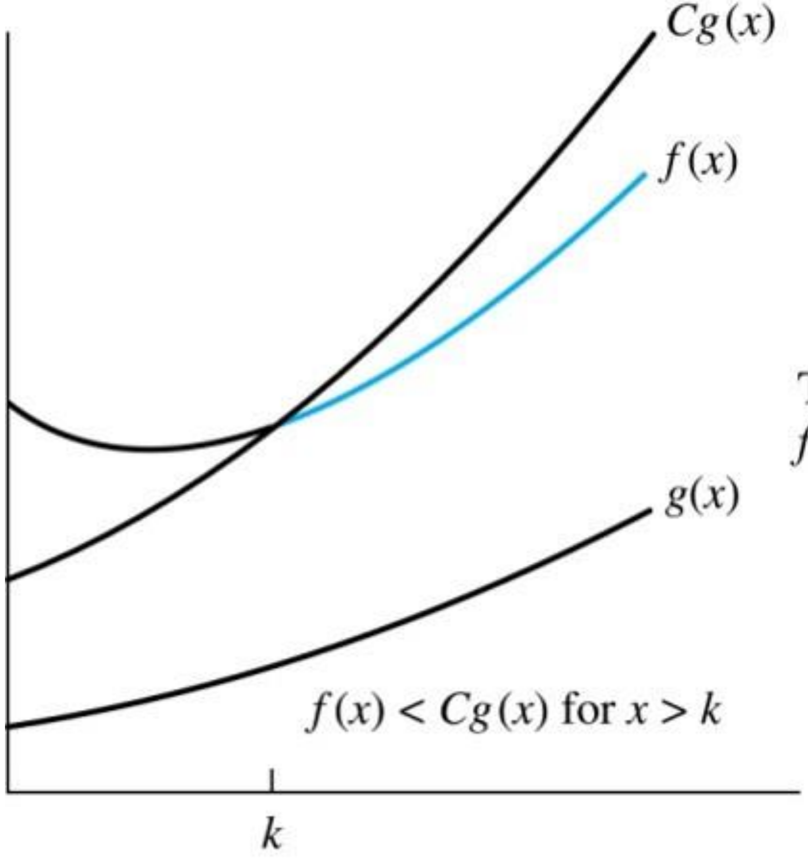
$$|f(x)| \leq C|g(x)|$$

whenever $x > k$. (illustration on next slide)

This is read as "*f*(*x*) is big-*O* 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.

Illustration of Big-*O* Notation 1



$$f\left(x\right) \text{ is } O\left(g\left(x\right)\right)$$

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

Jump to long description

Some Important Points 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 $|f(x)| \leq C|g(x)|$.

- Any pair C' and k' where $C < C'$ and $k < k'$ is also a pair of witnesses since whenever $x > k' > k$. $|f(x)| \leq C|g(x) \leq C'|g(x)|$

You may see “ *f*(*x*) = *O*(*g*(*x*))” instead of “ *f*(*x*) is *O*(*g*(*x*)).”

- But this is an abuse of the equals sign since the meaning is that there is an inequality relating the values of *f* and *g*, for sufficiently large values of x.

- It is ok to write *f*(*x*) ∊ *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.

Using the Definition of Big-*O* Notation 1

Example: Show that $|f(x)| \leq C|g(x)|$ $O(x^2)$ is

.

Solution: Since when $x > 1$, $x < x^2$ and $1 < x^2$

$$0 \leq x^2 + 2x + 1 \leq x^2 + 2x^2 + x^2 = 4x^2$$

- Can take $C = 4$ and $k = 1$ as witnesses to show that

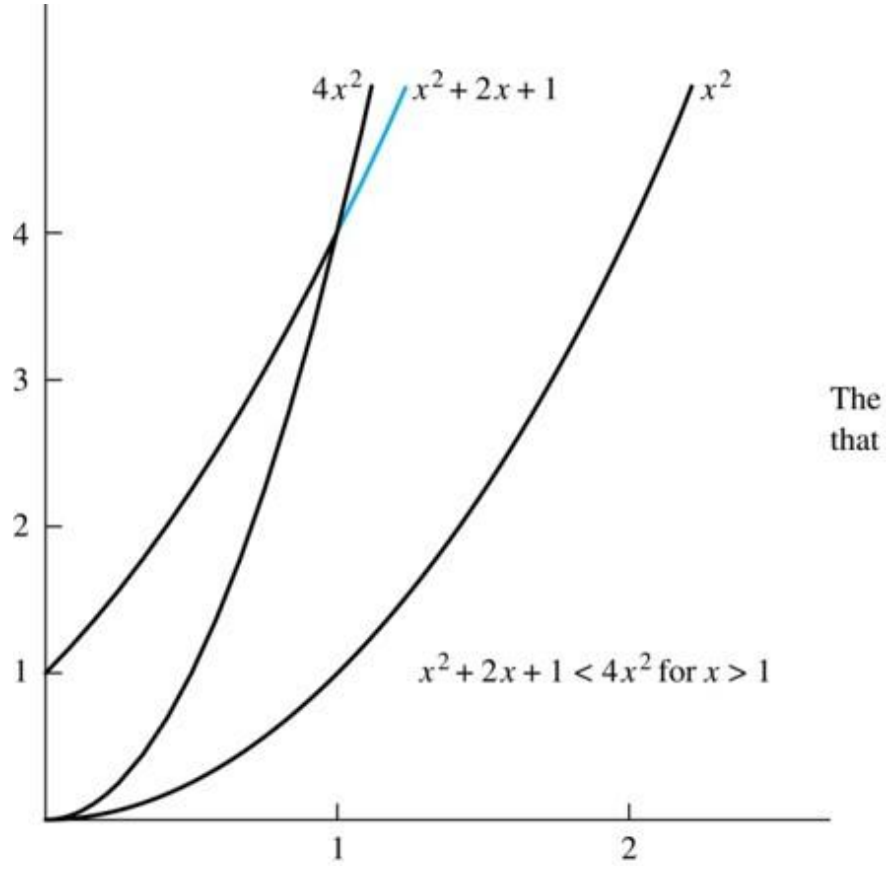
(see graph on next slide)

Alternatively, when $x > 2$, we have $2x \leq x^2$ and $1 < x^2$. Hence, $0 \leq x^2 + 2x + 1 \leq x^2 + x^2 + x^2 = 3x^2$ when $x > 2$.

- Can take $C = 3$ and $k = 2$ as witnesses instead.

Illustration of Big-*O* Notation 2

$$f(x) = x^2 + 2x + 1 \text{ is } \quad O(x^2)$$



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

Jump to long description

Big-*O* Notation 2

Both $f(x) = x^2 + 2x + 1$ and $g(x) = x^2$
are such that $f(x)$ is $O(g(x))$ and $g(x)$ is $O(f(x))$.
We say that the two functions are of the *same order*. (More on this later)

If $f(x)$ is $O(g(x))$ and *h(x)* is larger than *g(x)* for all positive real numbers, then $f(x)$ is $O(h(x))$.

Note that if $|f(x)| \leq C\,|g(x)|$ for $x > k$ and if $|h(x)| > |g(x)|$ for all *x*, then $|f(x)| \leq C\,|h(x)|$ if $x > k$. Hence, $f(x)$ is $O(h(x))$.

For many applications, the goal is to select the function *g*(*x*) in *O(g(x))* as small as possible (up to multiplication by a constant, of course).

Using the Definition of Big-*O* Notation 2

Example: Show that $7x^2$ is $O(x^3)$.

Solution: When $x > 7$, $7x^2 < x^3$. Take $C = 1$ and $k = 7$ as witnesses to establish that $7x^2$ is $O(x^3)$.

(Would $C = 7$ and $k = 1$ work?)

Example: Show that n^2 is not $O(n)$.

Solution: Suppose there are constants C and k for which $n^2 \leq Cn$, whenever $n > k$.
Then (by dividing both sides of $n^2 \leq Cn$) by n, then $n \leq C$ must hold for all $n > k$. A contradiction!

Big-*O* Estimates for Polynomials

Example: Let $f(x) = a_n x^n + a_{n-1}x^{n-1} + \cdots + a_1 x + a_0$

where $a_0, a_1, \ldots, a_n$ are real numbers with $a_n \neq 0$. Then $f(x)$ is $O(x^n)$.

Uses triangle inequality,
an exercise in Section 1.8.

Proof:
$$\begin{aligned} |f(x)| &= |a_n x^n + a_{n-1}x^{n-1} + \cdots + a_1 x^1 + a_0| \\ &\leq |a_n|x^n + |a_{n-1}|x^{n-1} + \cdots + |a_1|x^1 + |a_0| \\ &= x^n\left(|a_n| + |a_{n-1}|/x + \cdots + |a_1|/x^{n-1} + |a_0|/x^n\right) \\ &\leq x^n\left(|a_n| + |a_{n-1}| + \cdots + |a_1| + |a_0|\right) \end{aligned}$$

Assuming $x > 1$

Take $C = |a_n| + |a_{n-1}| + \cdots + |a_0|$ and $k = 1$. Then $f(x)$ is $O(x^n)$.

The leading term $a_n x^n$ of a polynomial dominates its growth.

Big-*O* Estimates for some Important Functions 1

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

Solution:

$$1 + 2 + \mathrm{L}\ + n \ \leq \ n + n + \mathrm{L}\ n = n^2$$

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

Example: Use big-*O* notation to estimate the factorial function $f\left(n\right) = n! = 1 \times 2 \times \mathrm{L}\ \times n$.

Solution:

$$n! = 1 \times 2 \times \mathrm{L}\ \times n \ \leq \ n \times n \times \mathrm{L}\ \times n = n^n$$

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

Big-*O* Estimates for some Important Functions 2

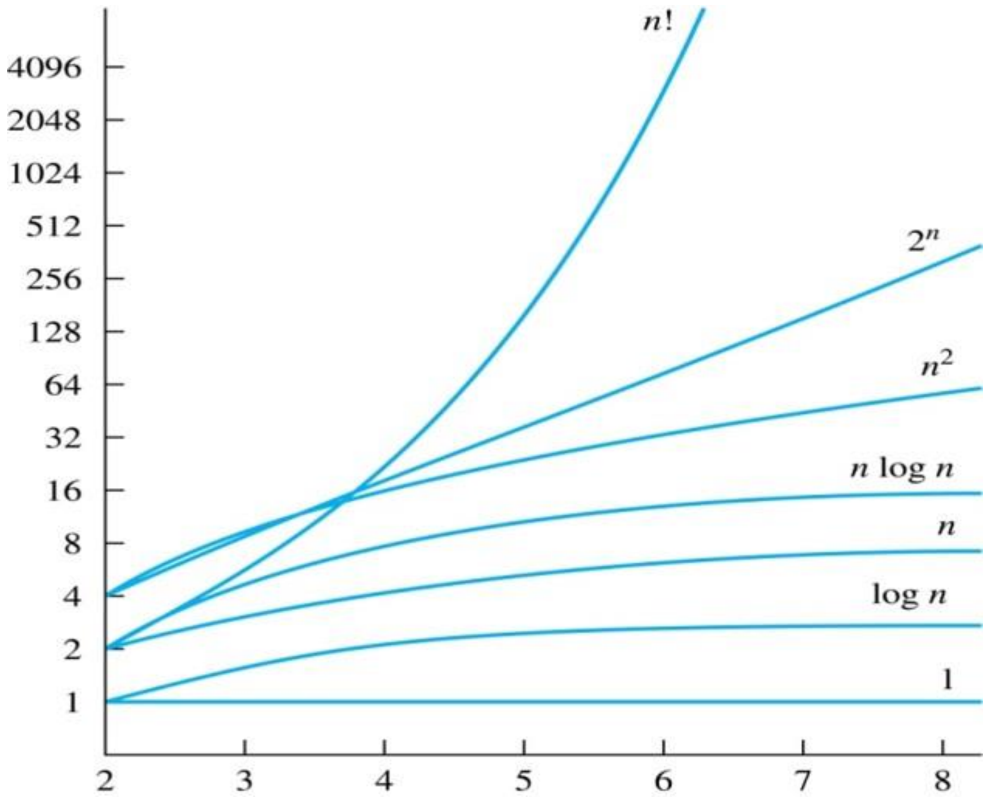
Example: Use big-*O* notation to estimate log *n*!

Solution: Given that $n! \leq n^n$ (previous slide)

then $\log(n!) \leq n \cdot \log(n)$.

Hence, log(*n*!) is *O*(*n*·log(*n*)) taking *C* = 1 and *k* = 1.

Display of Growth of Functions



Note the difference in behavior of functions as *n* gets larger

Jump to long description

Useful Big-*O* Estimates Involving Logarithms, Powers, and Exponents

If $d > \mathrm{c} > 1$, then

n^c is $O(n^d)$, but n^d is not $O(n^c)$.

If $\mathrm{b} > 1$ and c and d are positive, then

$(\log_{\mathrm{b}} n)^c$ is $O(n^d)$, but n^d is not $O((\log_{\mathrm{b}} n)^c)$.

If $\mathrm{b} > 1$ and d is positive, then

n^d is $O(b^n)$, but b^n is not $O(n^d)$.

If $c > \mathrm{b} > 1$, then

b^n is $O(c^n)$, but c^n is not $O(b^n)$.

Combinations of Functions[1]

If $f_1(x)$ is $O(g_1(x))$ and $f_2(x)$ is $O(g_2(x))$ then

$$(f_1 + f_2)(x) \text{ is } O(\max(|g_1(x)|, |g_2(x)|)).$$

- See next slide for proof

If $f_1(x)$ and $f_2(x)$ are both $O(g(x))$ then

$$(f_1 + f_2)(x) \text{ is } O(g(x)).$$

- See text for argument

If $f_1(x)$ is $O(g_1(x))$ and $f_2(x)$ is $O(g_2(x))$ then

$$(f_1 f_2)(x) \text{ is } O(g_1(x)g_2(x)).$$

- See text for argument

Combinations of Functions 2

If $f_1(x)$ is $O(\mathrm{g}_1(\mathrm{x}))$ and $f_2(x)$ is $O(\mathrm{g}_2(\mathrm{x}))$ then

$$(f_1 + f_2)(x) \text{ is } O(\max(|\mathrm{g}_1(x)|,|\mathrm{g}_2(\mathrm{x})|)).$$

- By the definition of big-*O* notation, there are constants C_1, C_2, k_1, k_2 such that $|f_1(x) \le C_1 |g_1(\mathrm{x})|$ when $\mathrm{x} > k_1$ and $f_2(x) \le C_2 | g_2(\mathrm{x})|$ when $x > k_2$.

$$|(f_1 + f_2)(x)| = |f_1(x) + f_2(x)| \quad \text{by the triangle inequality } |a + b| \le |a| + |b|$$

$$\le |f_1(x)| + |f_2(x)|$$

$$|f_1(x)| + |f_2(x)| \le C_1 |g_1(\mathrm{x})| + C_2 |g_2(\mathrm{x})|$$

$$\le C_1 |g(\mathrm{x})| + C_2 |g(\mathrm{x})| \quad \text{where } g(x) = \max(|\mathrm{g}_1(\mathrm{x})|,|\mathrm{g}_2(\mathrm{x})|)$$

$$= (C_1 + C_2) |g(x)|$$

$$= C |g(x)| \quad \text{where } C = C_1 + C_2$$

- Therefore $|(f_1 + f_2)(x)| \le C |g(x)|$ whenever $x > k$, where $k = \max(k_1, k_2)$.

Ordering Functions by Order of Growth

Put the functions below in order so that each function is big-O of the next function on the list.

$f_1(n) = (1.5)^n$

$f_2(n) = 8n^3 + 17n^2 + 111$

$f_3(n) = (\log \mathrm{n})^2$

$f_4(n) = 2^n$

$f_5(n) = \log\ (\log n)$

$f_6(n) = n^2 (\log n)^3$

$f_7(n) = 2^n (n^2 + 1)$

$f_8(n) = n^3 + n(\log n)^2$

$f_9(n) = 10000$

$f_{10}(n) = n!$

We solve this exercise by successively finding the function that grows slowest among all those left on the list.

$f_9(n) = 10000$ (constant, does not increase with n)

$f_5(n) = \log(\log n)$ (grows slowest of all the others)

$f_3(n) = (\log \mathrm{n})^2$ (grows next slowest)

$f_6(n) = n^2 (\log n)^3$ (next largest,$(\log n)^3$ factor smaller than any power of n)

$f_2(n) = 8n^3 + 17n^2 + 111$ (tied with the one below)

$f_8(n) = n^3 + n(\log n)^2$ (tied with the one above)

$f_1(n) = (1.5)^n$ (next largest, an exponential function)

$f_4(n) = 2^n$ (grows faster than one above since 2 > 1.5)

$f_7(n) = 2^n (n^2 + 1)$ (grows faster than above because of the n^2 +1 factor)

$f_{10}(n) = n!$ ($n!$ grows faster thancn for every c)