

Discrete Structures for Computing

CSCE 222

Sandeep Kumar

Many slides based on [Lee19], [Rog21], [GK22]

Algorithms

Chapter 3

©2019 McGraw-Hill Education. All rights reserved. Authorized only for instructor use in the classroom. No reproduction or further distribution permitted without the prior written consent of McGraw-Hill Education.

Chapter Summary

Algorithms

- Example Algorithms
- Algorithmic Paradigms

Growth of Functions

- Big- O and other Notations

Complexity of Algorithms

Section Summary

- Properties of Algorithms
- Algorithms for Searching and Sorting
- Greedy Algorithms
- Halting Problem

Problems and Algorithms

- Many domains have key 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

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

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

- Set the temporary maximum equal to the first integer in the sequence.
- 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.
- Repeat the previous step if there are more integers. If not, stop.
- 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 used in the book is specified in Appendix 3.
 - ▶ It uses 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( $a_1, a_2, \dots, a_n$  : integer)
max :=  $a_1$ 
for  $i$  := 2 to  $n$ 
    if max <  $a_i$  then max :=  $a_i$ 
return max
```

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.

- *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, \dots, 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

The linear search algorithm locates an item in a list by examining elements in the sequence one at a time, starting at the beginning. See [Linear Search Animation](#).

- 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,  $a_1, a_2, \dots, a_n$ : 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 {subscript that equals  $x$ , or 0 if  $x$  not found }
```

Binary Search

Assume the input is a list of items in increasing order. [Animation](#).

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

```
procedure binary search( $x$ : int,  $a_1, a_2, \dots, a_n$ : increasing int)
   $i := 1$  { $i$  is left endpoint of search interval}
   $j := n$  { $j$  is right endpoint of search 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

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

- 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

- 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

Binary Search

- The midpoint of the current list is now the 14th position with a value of 19. Since $19 \neq 19$, 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

- 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

- 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.

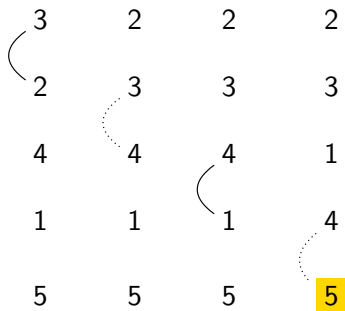
Bubble Sort

Bubble sort makes multiple passes through a list. Every pair of elements that are found to be out of order are interchanged. See [animation](#).

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

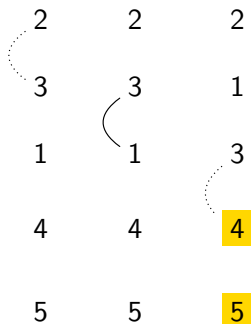
Bubble Sort. . .first pass

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



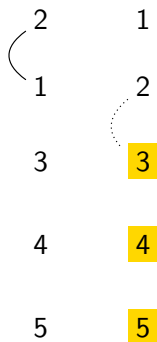
- At the first pass the largest element is put into the correct position

Bubble Sort. . .second pass



- At the end of the second pass, the 2nd largest element is put into the correct position.
- In each subsequent pass, the next largest element is put in the correct position.

Bubble Sort. . .third pass



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

Bubble Sort. . .fourth pass



Insertion Sort

Insertion sort begins with the 2^{nd} element. [Animation](#).

- 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( $a_1 \dots a_n$ : 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, \dots, a_n is in increasing order}

Insertion Sort. . .

Show all the steps of insertion sort with the input:

3, 2, 4, 1, 5

- 2, 3, 4, 1, 5 (first two positions are interchanged).
- 2, 3, 4, 1, 5 (third element remains in its position).
- 1, 2, 3, 4, 5 (fourth is placed at beginning).
- 1, 2, 3, 4, 5 (fifth element remains in its position).

Greedy Algorithms

Greed is good. Greed is right. Greed works. —'Wall Street'

Optimization problems minimize or maximize some objective function over all possible solutions. 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 Algorithms. . .

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

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

At each step choose the coin with the largest possible value that does not exceed the amount of change left. See [demo](#).

- If $n = 67¢$, first choose 2 quarters leaving $67 - 50 = 17¢$.
- Then choose 1 dime, leaving $17 - 10 = 7¢$.
- Choose 1 nickel, leaving $7 - 5 = 2¢$.
- Choose 2 pennies.

Greedy Change-Making Algorithm

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

```
procedure change( $c_1, c_2, \dots, c_r$ : coin values,  
                 $c_1 > c_2 > \dots > c_r; n > 0$ )  
  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 $c_1 = 25, c_2 = 10, c_3 = 5, c_4 = 1$.

Proving Optimality for U.S. Coins

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¢; 2 dimes and 4 pennies.
 - ▶ 2 dimes, 0 nickels, 4 pennies = 24¢.
 - ▶ 1 dime, 1 nickel, 4 pennies = 19¢.

Proving Optimality for U.S. Coins

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

Proof: By contradiction. 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.

- But $q' \leq q$ where q' is the number of quarters used in this optimal way and q is the number of quarters in the greedy algorithm's solution. Because q is the max # of quarters possible with n cents.

But if that was true, then you'd have 25¢ to make out of dimes, nickels and pennies. And that is not possible by Lemma 1.

- Similarly, by Lemma 1, the two algorithms must have the same number of dimes, nickels, and quarters.
 - ▶ If $d' < d$, then we'd need to make change for 10¢ out of nickels and pennies, but with only 4 pennies possible we'd need 2 nickles which is not possible.
 - ▶ Else, using dynamic programming one can show that greedy is optimal for $n < 25$.

Greedy Change-Making Algorithm

- Optimality depends on the denominations available.
- For U.S. coins, optimality still holds if we add half dollar coins (50¢) and dollar coins (100¢).
- But if we allow only quarters (25¢), dimes (10¢), and pennies (1¢), the algorithm no longer produces the minimum number of coins.
 - ▶ Consider the example of 31¢. The optimal number of coins is 4, i.e., 3 dimes and 1 penny. What does the algorithm output?

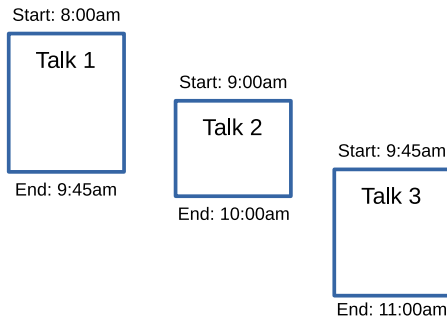
Greedy Scheduling

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. See [instance generator](#).

- 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

Picking the shortest talk doesn't work.



- Can you find a counterexample here?
- But picking the one that ends soonest *does* work.

Greedy Scheduling algorithm

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

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

Will be proven correct by induction in Chapter 5. Also See Section 4.2 of [Eri19].

Proof sketch—Greedy Scheduling algorithm

FYIO

See [Eri19, Section 4.2].

- Assume that greedy scheduling works for a set of intervals of size n .
- We need to show that greedy scheduling also works for sets of intervals of size $(n + 1)$.
 - ▶ By Lemma 4.3, at least one maximal conflict-free schedule includes the interval that finishes first.
 - ▶ If we remove this interval and all intervals that conflict with it, we are left with a set of size $\leq n$.
 - ▶ By the induction hypothesis, greedy scheduling works for it.
 - ▶ Therefore, greedy scheduling works for sets of size $(n + 1)$.
- Lemma 4.3: At least one maximal conflict-free schedule includes the class that finishes first.
 - ▶ *In any schedule, one can replace the first interval by the interval that finishes first and preserve the size of the set.*

Halting Problem

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?

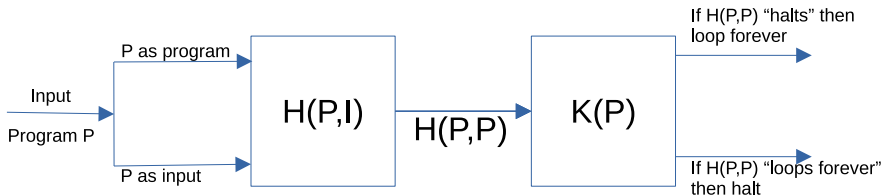
No. Proof by contradiction. Assume that there is such a procedure and call it $H(P, I)$. The procedure $H(P, I)$ takes as input a program P and the input I to P .

- H outputs “halt” if P will stop when run with input I .
- Otherwise, H outputs “loops forever”.

Halting Problem...

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.



$H(P, In) := P$ halts on input In ? *true* : *false*

$K(P) := H(P, P)$? *loop* : *halt*

Does $K(K)$ halt or loop?

Halting Problem...

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

Reminiscent of: The barber is the “one who shaves all those, and those only, who do not shave themselves”. The question is, does the barber shave himself?

Section Summary

- Big- O Notation
- Big- O Estimates for Important Functions
- Big- Ω and Big- Θ Notation

The Growth of Functions

In both computer science and 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.

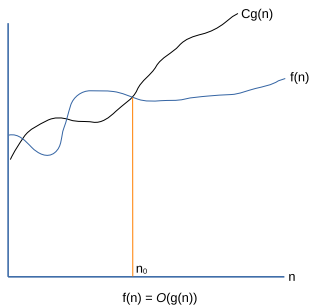
- That helps compare the efficiency of two different algorithms for solving the same problem.
- That also helps determine whether it is practical to use a particular algorithm as the input grows.
- We'll study these questions in Section 3.3.

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 $x > k$.

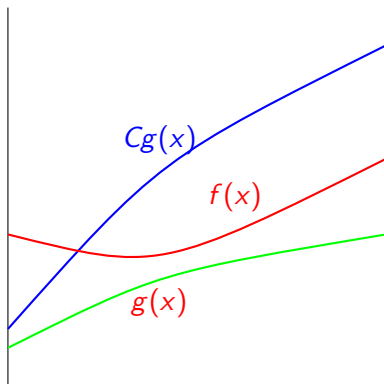


Big- O Notation

- 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.
- `plot [0:100] x*x+2*x+1, x*x.`
- `plot [0:100] x, x*x.`
- `plot [0:100] x, x*log(x), x*x.`
- `plot [0:100] x*x, 4*x*x+8*x+4.`

Illustration of Big- O Notation

- It appears that $g(x) < f(x)$ i.e., $g(x)$ is $O(f(x))$.
- But, $f(x)$ is also $O(g(x))$ because $f(x) < C \cdot g(x)$ for $x > k$.



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 $|f(x)| \leq C|g(x)| \leq C'|g(x)|$ whenever $x > k' > k$.

You may see “ $f(x) = O(g(x))$ ” instead of “ $f(x)$ is $O(g(x))$ ”. **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 $f(x) \in O(g(x))$, because $O(g(x))$ represents the set of functions that are $O(g(x))$.

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

Using the Definition of Big-O Notation

Show that $|f(x) = x^2 + 2x + 1| \leq C|g(x) = x^2|$ and is $\therefore O(x^2)$.

- 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.

- Alternatively, when $x > 2$, we have $2x \leq x^2$ and $1 < x^2$. Hence, **X**

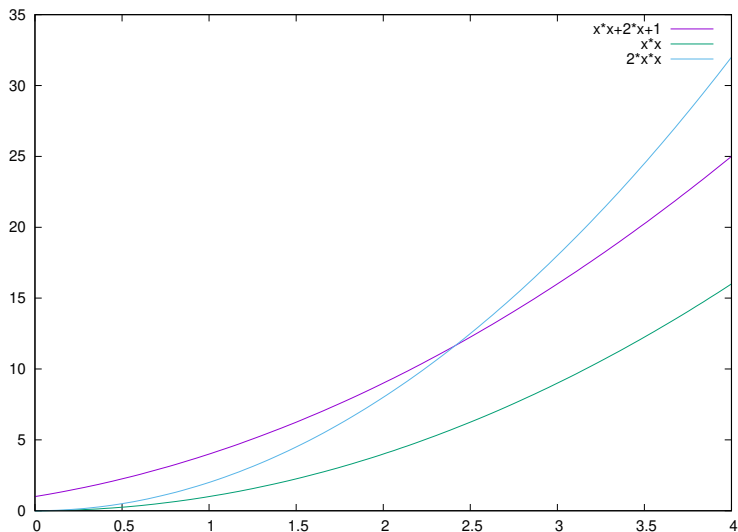
$$0 \leq x^2 + 2x + 1 \leq x^2 + x^2 + x^2 = 3x^2$$

when $x > 2$.

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

Illustration of Big-O Notation...

$f(x) = x^2 + 2x + 1$ is $O(x^2)$



Big-O Notation

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. If **X**

- $f(x)$ is $O(g(x))$, and
- $h(x) > g(x), x > 0$, then
- $f(x)$ is $O(h(x))$.

IOW, 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

- Show that $7x^2$ is $O(x^3)$.
 - ▶ When $x > 7$, $7x^2 < x^3$. Take $C = 1$ and $k = 7$ as witnesses to establish that $7x^2$ is $O(x^3)$.
- Show that n^2 is not $O(n)$.
 - ▶ Suppose there are constants C and k for which $n^2 \leq Cn$, whenever $n > k$.
 - ▶ Then dividing both sides by n , $n \leq C$ must hold for all $n > k$. A contradiction!

Big- O Estimates for Polynomials

Let

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

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

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

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

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

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

Thus, $1 + 2 + \cdots + n$ is $O(n^2)$ taking $C = 1$ and $k = 1$

- Use big- O notation to estimate the factorial function $n!$.

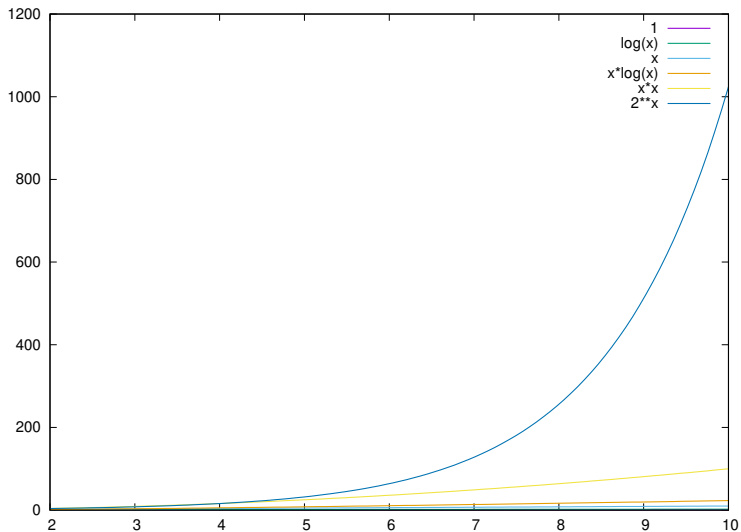
$$n! = 1 \times 2 \times \cdots \times n \leq n \times n \times \cdots \times n = n^n$$

Thus, $n!$ is $O(n^n)$ taking $C = 1$ and $k = 1$.

- Use big- O notation to estimate $\log n!$.

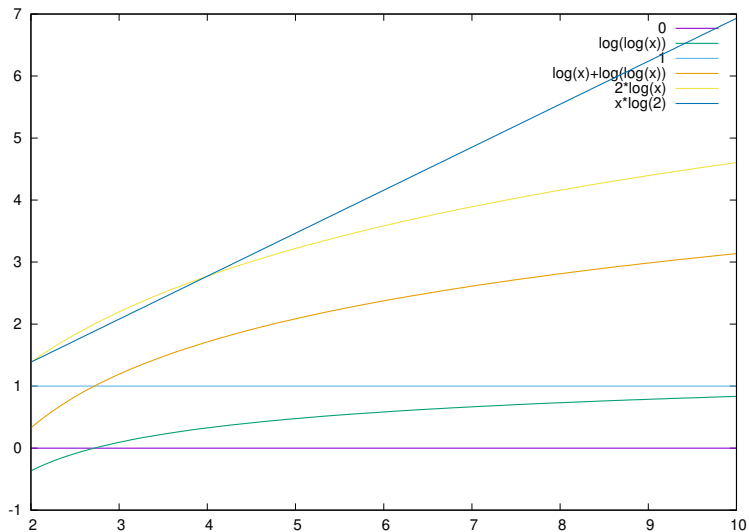
Given that $n! \leq n^n$, then $\log n! \leq n \log n$. Hence, $\log n!$ is $O(n \log n)$ taking $C = 1$ and $k = 1$.

Display of Growth of Functions



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

Display of Growth of Functions



y-axis is on a log scale.

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

- If $d > c > 1$, then

$$n^c \text{ is } O(n^d), \text{ but } n^d \text{ is not } O(n^c)$$

(Their difference $n^c(n^{d-c} - 1)$ cannot be expressed as $c \cdot n^c$.)

- If $b > 1$ and c and d are positive, then

$$(\log_b n)^c \text{ is } O(n^d), \text{ but } n^d \text{ is not } O((\log_b n)^c)$$

(Any power of log is asymptotically smaller than any polynomial).

- If $b > 1$ and d is positive, then

$$n^d \text{ is } O(b^n), \text{ but } b^n \text{ is not } O(n^d)$$

(Any polynomial is smaller than an exponential).

- If $c > b > 1$, then

$$b^n \text{ is } O(c^n), \text{ but } c^n \text{ is not } O(b^n)$$

(Two exponentials with different bases are different).

Combinations of Functions

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

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

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

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

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

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

Combinations of Functions...~~X~~

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

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

By the definition of big- O notation, there are constants C_1, C_2, k_1, k_2 such that

$$f_1(x) \leq C_1|g_1(x)|, x > k_1, \text{ and } f_2(x) \leq C_2|g_2(x)|, x > k_2$$

$$\begin{aligned} |(f_1 + f_2)(x)| &= |f_1(x) + f_2(x)| \\ &\leq |f_1(x)| + |f_2(x)| \\ |f_1(x)| + |f_2(x)| &\leq C_1|g_1(x)| + C_2|g_2(x)| \\ &\leq C_1|g(x)| + C_2|g(x)| \\ &= (C_1 + C_2)|g(x)| \\ &= C|g(x)| \end{aligned}$$

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.

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

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

$$f_3(n) = (\log n)^2$$

$$f_4(n) = 2^n$$

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

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

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

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

$$f_9(n) = 10000$$

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

Ordering Functions by Order of Growth

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

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

$$f_3(n) = (\log n)^2$$

$$f_4(n) = 2^n$$

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

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

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

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

$$f_9(n) = 10000$$

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

$$f_9(n) = 10000 \text{ (constant, does not increase with } n \text{)}$$

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

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

$$f_6(n) = n^2(\log n)^3 \text{ (next largest, } (\log n)^3 \text{ factor smaller than } n^2 \text{)}$$

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

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

$$f_1(n) = (1.5)^n \text{ (next largest, an exponential function)}$$

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

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

$$f_{10}(n) = n! \text{ (grows faster than } c^n \text{ for every } c \text{)}$$

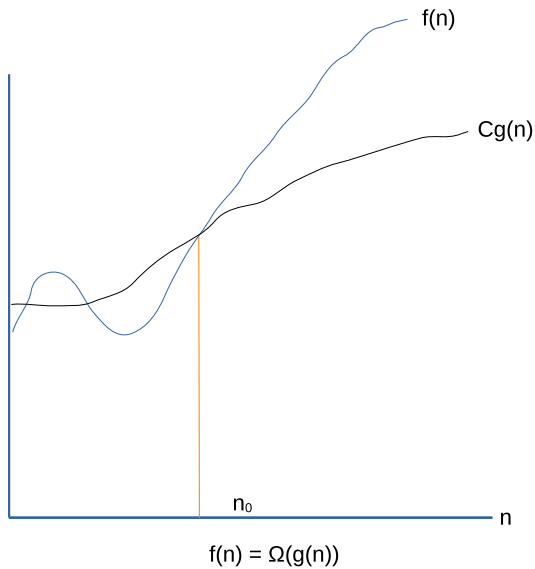
Big- Ω Notation

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

$$|f(x)| \geq C|g(x)|, x > k$$

- We say that “ $f(x)$ is big- Ω of $g(x)$ ”.
- Big- O gives an upper bound on the growth of a function, while Big- Ω gives a *lower bound*. Big- Ω 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 $O(f(x))$.
- Ω is the upper case version of the lower case Greek letter ω .

Big-Ω Notation



Big- Ω Notation

- Show that $f(x) = 8x^3 + 5x^2 + 7$ is $\Omega(x^3)$.
- $f(x) = 8x^3 + 5x^2 + 7 \geq x^3$ for all +ve real numbers x .
- Is it also the case that x^3 is $\Omega(8x^3 + 5x^2 + 7)$?
 - ▶ Yes. x^3 is $\Theta(8x^3 + 5x^2 + 7)$?

Big- Θ 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

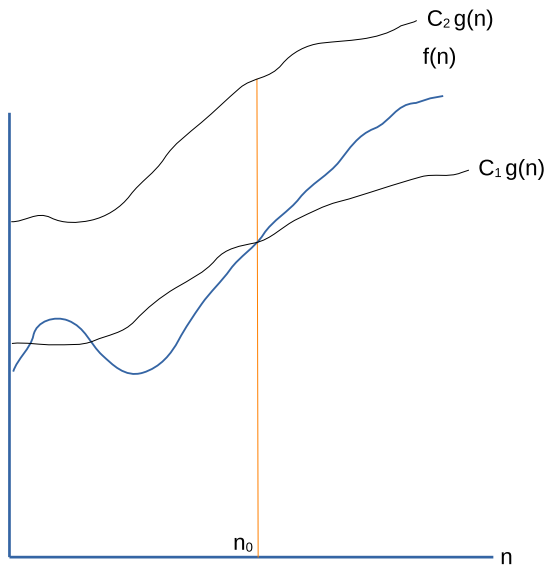
$f(x)$ is $\Theta(g(x))$ if $f(x)$ is $O(g(x))$ and $f(x)$ is $\Omega(g(x))$

- We say that “ f is big- Θ 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

$$C_1g(x) < f(x) < C_2g(x)$$

if $x > k$. This follows from the definitions of big- O and big- Ω .

Big- Θ Notation



$$f(n) = \Theta(g(n))$$

Big- Θ Notation

Show that the sum of the first n positive integers is $\Theta(n^2)$.

- We have already shown that $f(n)$ is $O(n^2)$.
- To show that $f(n)$ is $\Omega(n^2)$, we need a positive constant C such that $f(n) > Cn^2$ for sufficiently large n . Summing only the terms greater than $n/2$ we obtain the inequality

$$\begin{aligned}1 + 2 + \cdots + n &\geq \lceil n/2 \rceil + (\lceil n/2 \rceil + 1) + \cdots + n \\ &\geq \lceil n/2 \rceil + \lceil n/2 \rceil + \cdots + \lceil n/2 \rceil \\ &= (n - \lceil n/2 \rceil + 1)\lceil n/2 \rceil \\ &\geq (n/2)(n/2) = n^2/4\end{aligned}$$

- Taking $C = 1/4$, $f(n) > Cn^2$ for all positive integers n . Hence, $f(n)$ is $\Omega(n^2)$, and we can conclude that $f(n)$ is $\Theta(n^2)$.

Big- Θ Notation

Show that $f(x) = 3x^2 + 8x \log x$ is $\Theta(x^2)$.

- $3x^2 + 8x \log x \leq 11x^2$ for $x > 1$, since $0 \leq 8x \log x \leq 8x^2$.
- Hence, $3x^2 + 8x \log x$ is $O(x^2)$.
- x^2 is clearly $O(3x^2 + 8x \log x)$.
- Hence, $3x^2 + 8x \log x$ is $\Theta(x^2)$.

Big- Θ Notation

When $f(x)$ is $\Theta(g(x))$ it must also be the case that $g(x)$ is $\Theta(f(x))$.

Note that $f(x)$ is $\Theta(g(x))$ if and only if it is the case that

$$f(x) \text{ is } O(g(x)), \text{ and } g(x) \text{ is } O(f(x))$$

Sometimes writers are careless and write as if big- O notation has the same meaning as big- Θ .

Big- Θ Estimates for Polynomials

Theorem: Let

$$f(x) = a_n x^n + a_{n-1} x^{n-1} + \cdots + a_x + a_0$$

where a_0, a_1, \dots, a_n are real numbers with $a_n \neq 0$. Then

$f(x)$ is of order x^n (or $\Theta(x^n)$)

The proof is an exercise.

Examples:

- The polynomial $f(x) = 8x^5 + 5x^2 + 10$ is of order x^5 (or $\Theta(x^5)$).
- The polynomial $f(x) = 8x^{199} + 7x^{100} + x^{99} + 5x^2 + 25$ is of order x^{199} (or $\Theta(x^{199})$).

Section Summary

- Time Complexity
- Worst-Case Complexity
- Algorithmic Paradigms
- Understanding the Complexity of Algorithms

The Complexity of Algorithms

Given an algorithm, how efficient is this algorithm for solving a problem given input of a particular size? To answer this question, we ask:

- How much time does this algorithm use to solve a problem?
- How much computer memory does this algorithm use to solve a problem?

When we analyze the time the algorithm uses to solve the problem given input of a particular size, we are studying the *time complexity* of the algorithm.

When we analyze the computer memory the algorithm uses to solve the problem given input of a particular size, we are studying the *space complexity* of the algorithm.

The Complexity of Algorithms

- In this course, we focus on *time complexity*. The space complexity of algorithms is studied in later courses.
- We will measure time complexity in terms of the number of operations an algorithm uses and we will use big- O and big- Θ notation to estimate the time complexity.
- We can use this analysis to see whether it is practical to use this algorithm to solve problems with input of a particular size. We can also compare the efficiency of different algorithms for solving the same problem.
- We ignore implementation details (including the data structures used and both the hardware and software platforms) because it is extremely complicated to consider them.

Time Complexity

- To analyze the time complexity of algorithms, we determine the number of operations, such as comparisons and arithmetic operations (addition, multiplication, etc.). We can estimate the time a computer may actually use to solve a problem using the amount of time required to do basic operations.
- We ignore minor details, such as the “house keeping” aspects of the algorithm.
- We will focus on the *worst-case time* complexity of an algorithm. This provides an upper bound on the number of operations an algorithm uses to solve a problem with input of a particular size.
- It is usually much more difficult to determine the *average case time* complexity of an algorithm. This is the average number of operations an algorithm uses to solve a problem over all inputs of a particular size.

Complexity Analysis of Algorithms

Describe the time complexity of the algorithm for finding the maximum element in a finite sequence.

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

Count the number of comparisons.

- The $max < a_i$ comparison is made $n - 1$ times.
- Each time i is incremented, a test is made to see if $i \leq n$.
- One last comparison determines that $i > n$.
- Exactly $2(n - 1) + 1 = 2n - 1$ comparisons are made.

Hence, the time complexity of the algorithm is $\Theta(n)$.

Worst-Case Complexity of Linear Search

Determine the time complexity of the linear search algorithm.

```
procedure linear search( $x$ : integer,  $a_1, a_2, \dots, a_n$ : 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 {subscript that equals  $x$ , or 0 if  $x$  not found }
```

Count the number of comparisons.

- At each step two comparisons are made; $i \leq n$ and $x \neq a_i$.
- To end the loop, one comparison $i \leq n$ is made.
- After the loop, one more $i \leq n$ comparison is made.

If $x = a_i$, $2i + 1$ comparisons are used. If x is not on the list, $2n + 1$ comparisons are made and then an additional comparison is used to exit the loop. So, in the worst case $2n + 2$ comparisons are made. Hence, the complexity is $\Theta(n)$.

Average-Case Complexity of Linear Search

FYIO

Describe the average case performance of the linear search algorithm. (Although usually it is very difficult to determine average-case complexity, it is easy for linear search.)

Assume the element is in the list and that the possible positions are equally likely. By the argument on the previous slide, if $x = a_i$, the number of comparisons is $2i + 1$.

$$\begin{aligned}\frac{3 + 5 + 7 + \dots + (2n + 1)}{n} &= \frac{2(1 + 2 + 3 + \dots + n) + n}{2} \\ &= \frac{2 \left[\frac{n(n+1)}{2} \right]}{n} + 1 \\ &= n + 2\end{aligned}$$

Hence, the average-case complexity of linear search is $\Theta(n)$.

Worst-Case Complexity of Binary Search

Describe the time complexity of binary search in terms of the number of comparisons used.

Assume (for simplicity) $n = 2^k$ elements. Note that $k = \log n$.

- Two comparisons are made at each stage; $i < j$, and $x > a_m$.
- At the first iteration the size of the list is 2^k and after the first iteration it is 2^{k-1} . Then 2^{k-2} and so on until the size of the list is $2^1 = 2$.
- At the last step, a comparison tells us that the size of the list is $2^0 = 1$ and the element is compared with the single remaining element.
- Hence, at most $2k + 2 = 2 \log n + 2$ comparisons are made.
- Therefore, the time complexity is $\Theta(\log n)$, better than linear search.

Worst-Case Complexity of Bubble Sort ~~X~~

What is the worst-case complexity of bubble sort in terms of the number of comparisons made?

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

A sequence of $n - 1$ passes is made through the list. On each pass $n - i$ comparisons are made.

$$(n - 1) + (n - 2) + \dots + 2 + 1 = \frac{n(n - 1)}{2}$$

The worst-case complexity of bubble sort is $\Theta(n^2)$ since

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

Worst-Case Complexity of Insertion Sort ✗

What is the worst-case complexity of insertion sort in terms of the number of comparisons made?

The total number of comparisons are

$$2 + 3 + \dots + n = \frac{n(n-1)}{2} - 1$$

Therefore, the complexity is $\Theta(n^2)$.

```
procedure insertion sort( $a_1 \dots a_n$ :  
    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  $j - i - 1$   
             $a_{j-k} := a_{j-k-1}$   
             $a_i := m$ 
```

Algorithmic Paradigms

An *algorithmic paradigm* is a general approach based on a particular concept for constructing algorithms to solve a variety of problems.

- Greedy algorithms were introduced in Section 3.1.
- We discuss brute-force algorithms in this section.
- We will see
 - ▶ divide-and-conquer algorithms (Chapter 8),
 - ▶ dynamic programming (Chapter 8),
 - ▶ backtracking (Chapter 11), and
 - ▶ probabilistic algorithms (Chapter 7).
 - ▶ There are many other paradigms that you may see in later courses.

Brute-Force Algorithms

A *brute-force* algorithm is solved in the most straightforward manner, without taking advantage of any ideas that can make the algorithm more efficient.

Brute-force algorithms we have previously seen are sequential search, bubble sort, and insertion sort.

Computing the Closest Pair of Points by Brute-Force

Construct a brute-force algorithm for finding the closest pair of points in a set of n points in the plane and provide a worst-case estimate of the number of arithmetic operations.

Recall that the distance between (x_i, y_i) and (x_j, y_j) is

$$\sqrt{(x_j - x_i)^2 + (y_j - y_i)^2}$$

A brute-force algorithm simply computes the distance between all pairs of points and picks the pair with the smallest distance.

Note: There is no need to compute the square root, since the square of the distance between two points is smallest when the distance is smallest.

Computing the Closest Pair of Points by Brute-Force

Algorithm for finding the closest pair in a set of n points.

```
procedure closest pair( $(x_1, y_1), (x_2, y_2), \dots, (x_n, y_n)$ :  $x_i, y_i \in R$ )
  min =  $\infty$ 
  for i := 2 to n
    for j := 1 to i - 1
      if  $(x_j - x_i)^2 + (y_j - y_i)^2 < min$ 
        min :=  $(x_j - x_i)^2 + (y_j - y_i)^2$ 
        closest pair :=  $(x_i, y_i), (x_j, y_j)$ 
      endif
  return closest pair
```

The algorithm loops through $n(n-1)/2$ pairs of points, computes the value $(x_j - x_i)^2 + (y_j - y_i)^2$ and compares it with the minimum, etc. So, the algorithm uses $\Theta(n^2)$ arithmetic and comparison operations.

We will develop an algorithm with $O(n \log n)$ worst-case complexity in Section 8.3.

Understanding the Complexity of Algorithms

Complexity	Terminology
$\Theta(1)$	Constant complexity
$\Theta(\log n)$	Logarithmic complexity
$\Theta(n)$	Linear complexity
$\Theta(n \log n)$	Linearithmic complexity
$\Theta(n^b)$	Polynomial complexity
$\Theta(b^n), b > 1$	Exponential complexity
$\Theta(n!)$	Factorial complexity

Checkin.

- $2n$ is $O(n)$?
- n^2 is $O(n)$?
- n^2 is $O(n \log^2 n)$?
- $n \log n$ is $O(n^2)$?

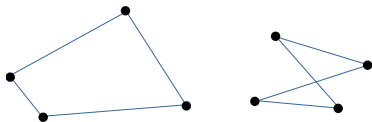
Understanding the Complexity of Algorithms

The Computer Time Used by Algorithms. Times of more than 10^{100} years are indicated with an *. 1 bit operation takes 10^{-11} s.

Size	Bit Operations Used					
n	$\log n$	n	$n \log n$	n^2	2^n	$n!$
10	$3 \times 10^{-11} s$	$10^{-10} s$	$3 \times 10^{-10} s$	$10^{-9} s$	$10^{-8} s$	$3 \times 10^{-7} s$
10^2	$7 \times 10^{-11} s$	$10^{-9} s$	$7 \times 10^{-9} s$	$10^{-7} s$	$4 \times 10^{11} y$	*
10^3	$1.0 \times 10^{-10} s$	$10^{-8} s$	$1 \times 10^{-7} s$	$10^{-5} s$	*	*
10^4	$1.3 \times 10^{-10} s$	$10^{-7} s$	$1 \times 10^{-6} s$	$10^{-3} s$	*	*
10^5	$1.7 \times 10^{-10} s$	$10^{-6} s$	$2 \times 10^{-5} s$	0.1s	*	*
10^6	$2 \times 10^{-10} s$	$10^{-5} s$	$2 \times 10^{-4} s$	0.17m	*	*

Complexity of Problems

- *Tractable Problem*: There exists a polynomial time algorithm to solve this problem. These problems are said to belong to the class P .
- *Intractable Problem*: There does not exist a polynomial time algorithm to solve this problem.
- *Unsolvable Problem*: No algorithm exists to solve this problem, e.g., the halting problem.
- *Class NP*: Solution can be checked in polynomial time. But no polynomial time algorithm has been found for finding a solution to problems in this class.
- *NP Complete Class*: If you find a polynomial time algorithm for one member of the class, it can be used to solve all the problems in the class.



P Versus NP Problem

The *P versus NP* problem asks whether the class $P = NP$? Are there problems whose solution can be **checked** in polynomial time, but which cannot be **solved** in polynomial time?

- Note that just because no one has found a polynomial time algorithm is different from showing that the problem cannot be solved by a polynomial time algorithm.

If a polynomial time algorithm for any of the problems in the NP complete class were found, then that algorithm could be used to obtain a polynomial time algorithm for every problem in the *NP* complete class.

- Satisfiability (in Section 1.3) is an *NP* complete problem.

Satisfiability Problem

- A *Boolean formula* ϕ has boolean variables and operations \wedge, \vee, \neg .
- ϕ is satisfiable if ϕ evaluates to True for some assignment to its variables. Sometimes we use 1 for True and 0 for False.
 - ▶ $\phi = (x \vee y) \wedge (\bar{x} \vee \bar{y})$ is satisfiable for $(x = 1, y = 0)$.
- $SAT = \{\langle \phi \rangle \mid \phi \text{ is a satisfiable Boolean formula}\}$.
 - ▶ Is $SAT \in NP$?
- **Theorem (Cook, Levin 1971):** $SAT \in P \rightarrow P = NP$.


P Versus NP Problem


It is generally believed that $P \neq NP$ since no one has been able to find a polynomial time algorithm for any of the problems in the NP complete class.


The problem of P versus NP remains one of the most famous unsolved problems in mathematics (including theoretical computer science). The Clay Mathematics Institute has offered a prize of \$1,000,000 for a solution.

Bibliography I

 Jeff Erickson.
Algorithms.
Independently published, 1st edition, June 2019.

 Ashutosh Gupta and S. Krishna.
Cs 228: Logic for computer science 2022.
<https://www.cse.iitb.ac.in/~akg/courses/2022-logic/>, January 2022.

 Hyunyoung Lee.
Discrete structures for computing.
Class slides for TAMU CSCE 222, 2019.

 Phillip Rogaway.
Ecs20 fall 2021 lecture notes, Fall 2021.