Algorithms

Chapter 3

Chapter Summary

- Algorithms
  - Example Algorithms
  - Algorithmic Paradigms
- Growth of Functions
  - Big-*O* and other Notation
- Complexity of Algorithms

Algorithms

Section 3.1

Section Summary

- 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

〖Example 1〗 Describe an algorithm for finding the maximum (largest) value in a finite sequence of integers.

***Solution* :**

$a_1 , a_2 , a_3 , \dots , a_n$

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, and 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 in the sequence.**
4. **Stop when there are no integers left in the sequence. The temporary maximum at this point is the largest integer in the sequence.**

9

Finding the Maximum Element in a Finite Sequence

- The algorithm in pseudocode:

```
procedure max(a1, a2, ...., an: integers)
   max := a1
   for i := 2 to n
      if max < ai then max := ai
   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,
          a1, a2, ...,an: distinct integers)
i := 1
while (i ≤ n and x ≠ ai)
    i := i + 1
if i ≤ 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

- 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

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

```
procedure binary search(x: integer, a_1,a_2,..., 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 := ⌊(i + j)/2⌋
      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

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 ≯ 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 18th position through the 18th.

   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 16 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.

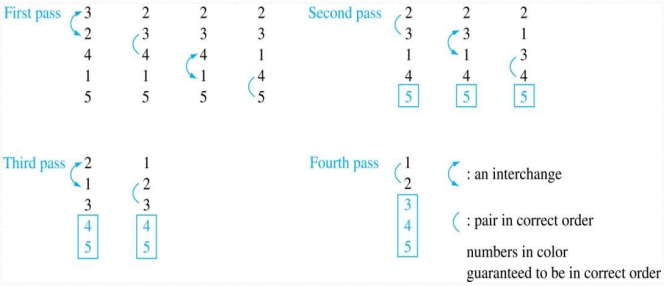
Bubble Sort

- *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

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.

Insertion Sort

- *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 $n \geq 2$)
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,...,a_n is in increasing order}

Insertion Sort

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) of n cents with the following 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

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

```
procedure change(c1, c2, ..., cr: values of coins, where c1> c2> ... > cr;
    n: a positive integer)
for i := 1 to r
    di := 0 [di counts the coins of denomination ci]
    while n ≥ ci
        di := di + 1 [add a coin of denomination ci]
        n = n - ci
[di counts the coins ci]
```

- 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

- 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

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, $q' \leq q$ 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. 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 Change-Making Algorithm

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

A failure of the greedy algorithm

- In some (fictional) monetary system, "Asiarons" come in 1 Asiaron, 7 Asiaron, and 10 Asiaron coins
- Using a greedy algorithm to count out 15 Asiarons, you would get
  - A 10 Asiaron piece
  - Five 1 Asiaron pieces, for a total of 15 Asiarons
  - This requires six coins
- A better solution would be to use two 7 Asiaron pieces and one 1 Asiaron piece
  - This only requires three coins
- The greedy algorithm results in a solution, but not in an optimal solution

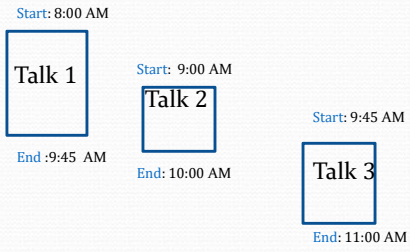
Greedy Scheduling

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

- Picking the shortest talk doesn't work.



- 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 ≤ s_2 ≤ ... ≤ s_n : start times, e_1 ≤ e_2 ≤ ... ≤ e_n :
    end times)
sort talks by finish time and reorder so that e_1 ≤ e_2 ≤ ... ≤ e_n
S := ∅
for j := 1 to n
   if talk j is compatible with S then
      S := S ∅∪{talk j}
return S [ S is the set of talks scheduled]
```

- Will be proven correct by induction in Chapter 5.

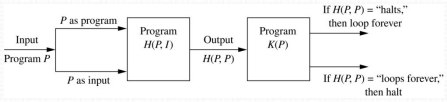
Halting Problem

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

- 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.



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

Halting Problem

```
Bool  DOES-HALT(P, I)
{
    if  (P will stop when run with input I)        //P(I) will stop
                return  1
     else
                return  0
}
Bool  SELF-HALT(program)
{ if(DOES-HALT(program, program))
        infinite loop
   else
        halt
}
```

Then how about

DOES-HALT(SELF-HALT, SELF-HALT)

Homework

Sec. 3.1 2, 4

The Growth of Functions

Section 3.2

Section Summary

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

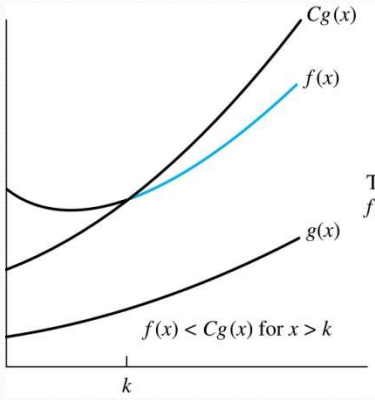
Definition: Let f and g be functions from the set of integers or the set of real numbers to the set of real numbers. We say that $f(x)$ is $O(g(x))$ if there are constants C and k such that

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

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*(凭证） to the relationship $f(x)$ is $O(g(x))$. Only one pair of witnesses is needed.

Illustration of Big-*O* Notation



$f(x)$ is $O(g(x))$

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

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 $O(g(x))$."

- 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 value sign since we will always deal with functions that take on positive values.

Using the Definition of Big-*O* Notation

Example: Show that $f(x) = x^2 + 2x + 1$ is $O(x^2)$.

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

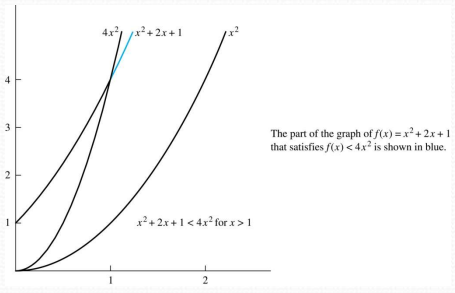
$$0 \leq x^2 + 2x + 1 \leq x^2 + 2x^2 + x^2 = 4x^2$$

- Can take $C = 4$ and $k = 1$ as witnesses to show that $f(x)$ is $O(x^2)$ (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

$f(x) = x^2 + 2x + 1$ 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*. (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

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

$\leq |a_n| x^n + |a_{n-1}| x^{n-1} + \cdots + |a_1| x^1 + |a_1|$

Assuming $x > 1$

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

$\leq x^n (|a_n| + |a_{n-1}| + \cdots + |a_1| + |a_1|)$

- Take $C = |a_n| + |a_{n-1}| + \cdots + |a_1| + |a_1|$ 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

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

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

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

Example: Use big-*O* notation to estimate the factorial function $f(n) = n! = 1 \times 2 \times \cdots \times n$.

Solution:

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

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

Continued →

Big-*O* Estimates for some Important Functions

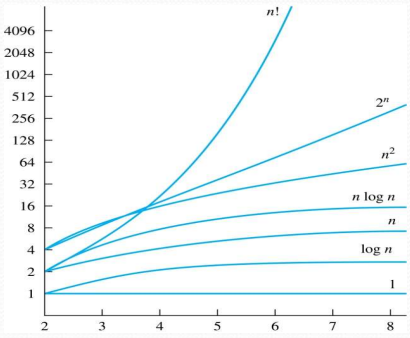
Example: Use big-*O* notation to estimate log *n*!

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

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

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

Display of Growth of Functions



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

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

- If $d > c > 1$, then
  n^c is $O(n^d)$, but n^d is not $O(n^c)$.
- If $b > 1$ and c and d are positive, then
  $(\log_b n)^c$ is $O(n^d)$, but n^d is not $O((\log_b n)^c)$.
- If $b > 1$ and d is positive, then
  n^d is $O(b^n)$, but b^n is not $O(n^d)$.
- If $c > b > 1$, then
  b^n is $O(c^n)$, but c^n is not $O(b^n)$.

Combinations of Functions

- If $f_1(x)$ is $O(g_1(x))$ and $f_2(x)$ is $O(g_2(x))$ then
  $(f_1 + f_2)(x)$ 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
  $(f_1 + f_2)(x)$ 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
  $(f_1 f_2)(x)$ is $O(g_1(x)g_2(x))$.
  - See text for argument

Combinations of Functions

- If $f_1(x)$ is $O(g_1(x))$ and $f_2(x)$ is $O(g_2(x))$ then
  $(f_1 + f_2)(x)$ 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 $|f_1(x) \le C_1|g_1(x)|$ when $x > k_1$ and $f_2(x) \le C_2|g_2(x)|$ when $x > k_2$.
  - $|(f_1 + f_2)(x)| = |f_1(x) + f_2(x)|$
    $\le |f_1(x)| + |f_2(x)|$ by the triangle inequality $|a + b| \le |a| + |b|$
  - $|f_1(x)| + |f_2(x)| \le C_1|g_1(x)| + C_2|g_2(x)|$
    $\le C_1|g(x)| + C_2|g(x)|$ where $g(x) = \max(|g_1(x)|,|g_2(x)|)$
    $= (C_1 + C_2)|g(x)|$
    $= C|g(x)|$ 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.
- $f_1(n) = (1.5)^n$
- $f_2(n) = 8n^3+17n^2 +111$
- $f_3(n) = (\log n)^2$
- $f_4(n) = 2^n$
- $f_5(n) = \log(\log n)$
- $f_6(n) = n^2(\log n)^3$
- $f_7(n) = 2^n(n^2 +1)$
- $f_8(n) = n^3 + n(\log n)^2$
- $f_9(n) = 10000$
- $f_{10}(n) = n!$

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)
- $f_3(n) = (\log n)^2$ (grows next slowest)
- $f_6(n) = n^2(\log n)^3$ (next largest, $(\log n)^3$ factor smaller than any power of n)
- $f_8(n) = n^3 + n(\log n)^2$ (tied with the one above)
- $f_2(n) = 8n^3+17n^2 +111$ (tied with the one below)
- $f_1(n) = (1.5)^n$ (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 than c^n for every c)

Big-Omega Notation

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 $\Omega(g(x))$ if there are constants *C* and *k* such that $|f(x)| \geq C|g(x)|$ when $x > k$.

Ω is the upper case version of the lower case Greek letter ω.

- We say that "*f*(*x*) is big-Omega of *g*(*x*)."
- 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. See the text for details.

Big-Omega Notation

Example: Show that $f(x) = 8x^3 + 5x^2 + 7$ is $\Omega(g(x))$ where $g(x) = x^3$.

Solution: $f(x) = 8x^3 + 5x^2 + 7 \geq 8x^3$ for all positive real numbers *x*.

- Is it also the case that $g(x) = x^3$ is $O(8x^3 + 5x^2 + 7)$?

Big-Theta Notation

Θ is the upper case version of the lower case Greek letter θ.

- **Definition**: 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-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.

Big Theta Notation

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

Solution: Let $f(n) = 1 + 2 + \cdots + n$.

- 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 $C = ¼$, $f(n) > Cn^2$ for all positive integers n. Hence, $f(n)$ is $\Omega(n^2)$, and we can conclude that $f(n)$ is $\Theta(n^2)$.

Big-Theta Notation

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

Solution:

- $3x^2 + 8x \log x \leq 11x^2$ 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-Theta 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 $f(x)$ is $O(g(x))$ and $g(x)$ is $O(f(x))$.
- Sometimes writers are careless and write as if big-O notation has the same meaning as big-Theta.

Big-Theta Estimates for Polynomials

Theorem: Let $f(x) = a_n x^n + a_{n-1} x^{n-1} + \cdots + a_1 x + a_o$ where $a_0, a_1, \ldots, a_n$ 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.)

Example:

The polynomial $f(x) = 8x^5 + 5x^2 + 10$ is order of x^5 (or $\Theta(x^5)$).

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

Homework

Sec. 3.2 8(c), 26(a), 54, 56

Complexity of Algorithms

Section 3.3

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-Theta 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

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

```
procedure max(a1, a2, ...., an: integers)
   max := a1
   for i := 2 to n
      if max < ai then max := ai
   return max{max is the largest element}
```

Solution: 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

Example: Determine the time complexity of the linear search algorithm.

```
procedure linear search(x:integer,
          a1, a2, ...,an: distinct integers)
i := 1
while (i ≤ n and x ≠ ai)
    i := i + 1
if i ≤ 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}
```

Solution: 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

Example: 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.)

Solution: 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 $2i + 1$.

$$\frac{3+5+7+\ldots+(2n+1)}{n} = \frac{2(1+2+3+\ldots+n)+n}{n} = \frac{2[\frac{n(n+1)}{2}]}{n} + 1 = n + 2$$

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

Worst-Case Complexity of Binary Search

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

```
procedure binary search(x: integer, a1,a2,..., an: increasing integers)
 i := 1 {i is the left endpoint of interval}
 j := n {j is right endpoint of interval}
 while i < j
      m := ⌊(i + j)/2⌋
      if x > am then i := m + 1
      else j := m
 if x = ai then location := i
 else location := 0
 return location{location is the subscript i of the term ai equal to x, or 0 if x is not found}
```

Solution: 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 the size 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

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

```
procedure bubblesort(a1,...,an: real numbers
                     with n ≥ 2)
   for i := 1 to n − 1
      for j := 1 to n − i
         if aj > aj+1 then interchange aj and aj+1
{a1,..., an is now in increasing order}
```

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

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

The worst-case complexity of bubble sort is $\Theta(n^2)$ since $\frac{n(n-1)}{2} = \frac{1}{2}n^2 - \frac{1}{2}n$.

Worst-Case Complexity of Insertion Sort

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

Solution: The total number of comparisons are:

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

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

```
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 − i − 1
         a_{j-k} := a_{j-k-1}
      a_i := m
```

Matrix Multiplication Algorithm

- The definition for matrix multiplication can be expressed as an algorithm; $\mathbf{C} = \mathbf{A}\,\mathbf{B}$ where $\mathbf{C}$ is an $m \times n$ matrix that is the product of the $m \times k$ matrix $\mathbf{A}$ and the $k \times n$ matrix $\mathbf{B}$.
- This algorithm carries out matrix multiplication based on its definition.

```
procedure matrix multiplication(A,B: matrices)
   for i := 1 to m
      for j := 1 to n
         c_ij := 0
         for q := 1 to k
            c_ij := c_ij + a_iq b_qj
return C{C = [c_ij] is the product of A and B}
```

$\mathbf{A} = [a_{ij}]$ is a $m \times k$ matrix
$\mathbf{B} = [b_{ij}]$ is a $k \times n$ matrix

Complexity of Matrix Multiplication

Example: How many additions of integers and multiplications of integers are used by the matrix multiplication algorithm to multiply two $n \times n$ matrices.

Solution: There are n^2 entries in the product. Finding each entry requires n multiplications and $n - 1$ additions. Hence, n^3 multiplications and $n^2(n - 1)$ additions are used.

Hence, the complexity of matrix multiplication is $O(n^3)$.

Boolean Product Algorithm

- The definition of Boolean product of zero-one matrices can also be converted to an algorithm.

```
procedure Boolean product(A,B: zero-one matrices)
   for i := 1 to m
      for j := 1 to n
         c_ij := 0
         for q := 1 to k
            c_ij := c_ij ∨ (a_iq ∧ b_qj)
return C{C = [c_ij] is the Boolean product of A and B}
```

Complexity of Boolean Product Algorithm

Example: How many bit operations are used to find $\mathbf{A} \odot \mathbf{B}$, where A and B are $n \times n$ zero-one matrices?

Solution: There are n^2 entries in the $\mathbf{A} \odot \mathbf{B}$. A total of n Ors and n ANDs are used to find each entry. Hence, each entry takes $2n$ bit operations. A total of $2n^3$ operations are used.

Therefore the complexity is $O(n^3)$

Matrix-Chain Multiplication

- How should the *matrix-chain* $\mathbf{A}_1\mathbf{A}_2 \cdots \mathbf{A}_n$ be computed using the fewest multiplications of integers, where $\mathbf{A}_1, \mathbf{A}_2, \cdots, \mathbf{A}_n$ are $m_1 \times m_2$, $m_2 \times m_3, \cdots m_n \times m_{n+1}$ integer matrices. Matrix multiplication is associative (exercise in Section 2.6).

Example: In which order should the integer matrices $\mathbf{A}_1\mathbf{A}_2\mathbf{A}_3$ - where $\mathbf{A}_1$ is 30×20 $\mathbf{A}_2$ 20×40, $\mathbf{A}_3$ 40×10 - be multiplied to use the least number of multiplications.

Solution: There are two possible ways to compute $\mathbf{A}_1\mathbf{A}_2\mathbf{A}_3$.

- $\mathbf{A}_1(\mathbf{A}_2\mathbf{A}_3)$: $\mathbf{A}_2\mathbf{A}_3$ takes $20 \cdot 40 \cdot 10 = 8000$ multiplications. Then multiplying $\mathbf{A}_1$ by the 20×10 matrix $\mathbf{A}_2\mathbf{A}_3$ takes $30 \cdot 20 \cdot 10 = 6000$ multiplications. So the total number is $8000 + 6000 = 14{,}000$.
- $(\mathbf{A}_1\mathbf{A}_2)\mathbf{A}_3$: $\mathbf{A}_1\mathbf{A}_2$ takes $30 \cdot 20 \cdot 40 = 24{,}000$ multiplications. Then multiplying the 30×40 matrix $\mathbf{A}_1\mathbf{A}_2$ by $\mathbf{A}_3$ takes $30 \cdot 40 \cdot 10 = 12{,}000$ multiplications. So the total number is $24{,}000 + 12{,}000 = 36{,}000$.

So the first method is best.

An efficient algorithm for finding the best order for matrix-chain multiplication can be based on the algorithmic paradigm known as *dynamic programming*. (see Ex. 57 in Section 8.1)

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

Example: 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.

Solution: 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.

Continued →

Computing the Closest Pair of Points by Brute-Force

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

```
procedure closest pair((x1, y1), (x2, y2), ... ,(xn, yn): xi, yi real numbers)
min = ∞
   for i := 1 to n
      for j := 1 to i
          if (xj − xi)² + (yj − yi)² < min
            then min := (xj − xi)² + (yj − yi)²
              closest pair := (xi, yi), (xj, yj)
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 $O(\log n)$ worst-case complexity in Section 8.3.

Understanding the Complexity of Algorithms

TABLE 1 Commonly Used Terminology for 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
$\Theta(b^n)$, where $b > 1$	Exponential complexity
$\Theta(n!)$	Factorial complexity

Understanding the Complexity of Algorithms

TABLE 2 The Computer Time Used by Algorithms.

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

Times of more than 10^{100} years are indicated with an *.

Example

- 以计算 nXn 阶行列式为例子。
- 如果我们用行列式定义去计算，其计算复杂度为 O(n•n!)
- 如果我们通过行消元将行列式化上三角行列计算，其计算复杂度为 $O(n^3)$
- 如果 n= 50,采用第一种方法所需要计算的乘法次数 $1.5X10^{65}$，第二种方法所需要的乘法次数为 $6X10^6$，如果用一台计算速度是每秒可执行1000亿次乘法的计算机计算，其计算时间分别为 $5x10^{46}$年及 $6x10^{-5}$秒

Complexity of Problems

- *Tractable*（易解） *Problem*: There exists a polynomial time algorithm to solve this problem. These problems are said to belong to the *Class P*.
- *Intractable*（难解） *Problem*: There does not exist a polynomial time algorithm to solve this problem
- *Unsolvable Problem* : No algorithm exists to solve this problem, e.g., 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

Stephen Cook
(Born 1939)

- The *P versus NP problem* asks whether the class P = NP? Are there problems whose solutions can be checked in polynomial time, but can not be solved in polynomial time?
  - Note that just because no one has found a polynomial time algorithm is different from showing that the problem can not 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.
- It is generally believed that P≠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.

Homework

Sec. 3.3 7, 10