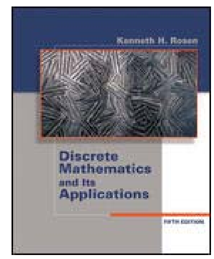
Chapter 2 (Part 1): The Fundamentals: Algorithms, the Integers & Matrices

- Algorithms (Section 2.1)
- The Growth of Functions (Section 2.2)
- Complexity of Algorithms (Section 2.3)



Algorithms (2.1)

- Some Applications:
  - Use of number theory to make message secret
  - Generate pseudorandom numbers
  - Assign memory locations to computer files
  - Internet security

Algorithms (2.1) (cont.)

- Introduction
  - Given a sequence of integers, find the largest one
  - Given a set, list all of his subsets
  - Given a set of integers, put them in increasing order
  - Given a network, find the shortest path between two vertices

Algorithms (2.1) (cont.)

- Methodology:
  - Construct a model that translates the problem into a mathematical context
  - Build a method that will solve the general problem using the model

  Ideally, we need a procedure that follows a sequence of steps that leads to the desired answer. Such a sequence is called an algorithm.

> *History:* the term algorithm is a corruption of the name Al-Khowarizmi (mathematician of the 9^{th} century)

Algorithms (2.1) (cont.)

– 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 largest value in a finite sequence of integers

Solution: We perform the following steps:

Algorithms (2.1) (cont.)

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 that 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

Pseudocode: intermediate step between an English language description of an algorithm and an implementation of this algorithm in a programming language

Algorithms (2.1) (cont.)

Algorithm: Finding the maximum element in a finite sequence

```
Procedure max(a1, a2, …, an: integer)
max := a1
For i := 2 to n
   If max < ai then max := ai
{max is the largest element}
```

Algorithms (2.1) (cont.)

– Properties of an algorithm:

- Input: an algorithm has input values from a specified set
- Output: from each set of input values an algorithm produces output values from a specified set. The output values are the solution to the problem
- Definiteness: the steps of an algorithm must be defined precisely
- Correctness: an algorithm should produce the correct output values for each set of input values
- Finiteness: an algorithm should produce the desired output after a finite (but perhaps large) number of steps for input in the set
- Effectiveness: it must be possible to perform each step of an algorithm exactly and in a finite amount of time
- Generality: the procedure should be applicable for all problems of the desired form not just for a particular set of input values.

Algorithms (2.1) (cont.)

- Searching Algorithms
  - Problem: "Locate an element x in a list of distinct elements $a_1, a_2, \ldots, a_n$, or determine that it is not in the list."

    We should provide as a solution to this search problem the location of the term in the list that equals x.

Algorithms (2.1) (cont.)

- The linear search

  Algorithm: 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
{location is the subscript of the term that equals
  x, or is 0 if x is not found}
```

Algorithms (2.1) (cont.)

- The binary search
  - Constraint: can be used when the list has terms occurring in order of increasing size (words listed in lexicographic order)
  - Methodology: Compare the element to be located to the middle term of the list

Algorithms (2.1) (cont.)

- Example: Search 19 in the list

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

  - split the list into 2 subsets with 8 terms each

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

  - Compare 19 with the largest element of the first set

    $10 < 19 \Rightarrow$ search 19 in the second set

  - Split the second subset into 2 smaller subsets

    12 13 15 16   18 19 20 22

  - Compare 19 with 16

    $16 < 19 \Rightarrow$ search 19 in the second set

  - Split the second subset as: 18 19   20 22
  - Compare $19 > 19$ is false $\Rightarrow$ search 19 in 18 19
  - Split the subset as : 18   19
  - Since $18 < 19 \Rightarrow$ search restricted to the second list
  - Finally 19 is located at the 14^{th} element of the original list

Algorithms (2.1) (cont.)

Algorithm: the binary search algorithm

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

Algorithms (2.1) (cont.)

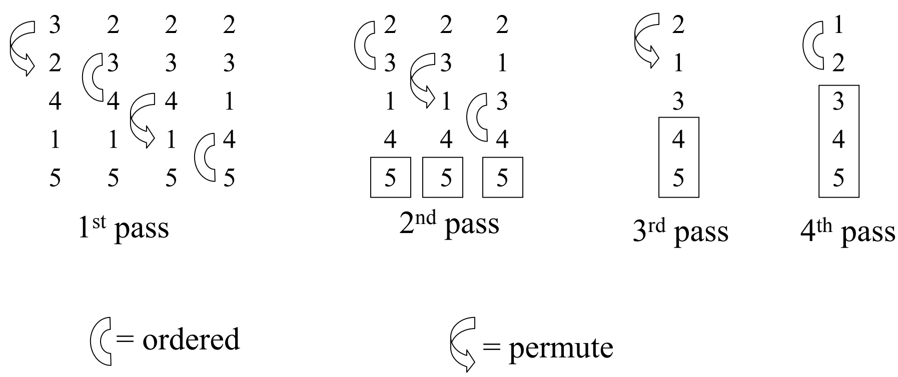
- Sorting
  - Goal:

    "Order the elements of a list". For example, sorting the list 7, 2, 1, 4, 5, 9 produces the list 1, 2, 4, , 5, 7, 9. Similarly, sorting the list d, h, c, a, f produces a, c, d, f, h.

– The Bubble sort

- Example: Sort the list 3, 2, 4, 1, 5 into increasing order using the Bubble sort

Steps of the Bubble sort



Algorithms (2.1) (cont.)

Algorithm: the Bubble sort

```
Procedure Bubblesort (a_1, …, a_n)
  for i := 1 to n-1 {count number of passes}
         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 the increasing order}
```

Algorithms (2.1) (cont.)

- Greedy algorithms
  - Goal: Solving optimization problems. Find a solution to the given problem that either minimizes or maximizes the value of some parameter
  - Some examples that involves optimization:
    - Find a route between 2 cities with smallest total mileage
    - Determine a way to encode messages using the fewest bits possible
    - Find a set of fiber links between networks nodes using the least amount of fiber

Algorithms (2.1) (cont.)

- The change making problem
  - Problem statement: Consider the problem of making n cents change with quarters, dimes, nickels and pennies, and using the <u>least total number of coins.</u>
  - For example, to make change for 67 cents, we do the following:
    1. Select a quarter, leaving 42 cents
    2. Select a second quarter, leaving 17 cents
    3. Select a dime, leaving 7 cents
    4. Select a nickel, leaving 2 cents
    5. Select a penny, leaving 1 cent
    6. Select a penny.

Algorithms (2.1) (cont.)

Algorithm: Greedy change making

```
Procedure change (c1, c2, …, cr: values of
  denominations of coins where c1 > c2 >…> cr; n:
  positive integer)
For i := 1 to r
  while n ≥ ci
   begin
         add a coin with value ci to the change
         n := n-ci
   end
```

Algorithms (2.1) (cont.)

- Remark: if we have only quarters, dimes and pennies ⇒ the change for 30 cents would be made using 6 coins = 1 quarter + 5 pennies.

  Whereas a <u>better</u> solution is equal to 3 coins = 3 dimes!

  Therefore:

  "The greedy algorithm selects the best choice at each step, instead of considering all sequences of steps that may lead to an optimal solution. The greedy algorithm often leads to a solution!"

Algorithms (2.1) (cont.)

- Lemma:

  If 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, at most 1 nickel, at most 4 pennies and cannot have 2 dimes and 1 nickel. The amount of change in dimes, nickels and pennies cannot exceed 24 cents.

Algorithms (2.1) (cont.)

- Proof of the lemma using contradiction:

  If we had more than the number of coins specified, then we will be able to replace them <u>using fewer coins</u> that have the same value.

  1. 3 dimes will be replaced by a quarter and 1 nickel
  2. 2 nickels replaced by a dime
  3. 5 pennies replaced by a nickel
  4. 2 dimes and I nickel replaced by a quarter

  Besides, we cannot have 2 dimes and 1 nickel $\Rightarrow$ 24 cents is the most money we can have in dimes, nickels and pennies when we make change using the fewest number of coins for n cents.

  Theorem [Greedy]: The greedy algorithm produces chab=nge using the fewest coins possible.

The Growth of Functions (Section 2.2)

- We quantify the concept that g grows at least as fast as f.
- What really matters in comparing the complexity of algorithms?
  - We only care about the behavior for large problems.
  - Even bad algorithms can be used to solve small problems.
  - Ignore implementation details such as loop counter incrementation, etc. We can straight-line any loop.

The Growth of Functions (2.2) (cont.)

- The Big-O Notation
  - **Definition:** Let f and g be functions from N to R.

    Then g asymptotically dominates f, denoted f is O(g) or 'f is big-O of g,' or 'f is order g,' iff

    $$\exists k\ \exists C\ \forall n\ [n > k \rightarrow |f(n)| \le C\ |g(n)|]$$
  - Note:
    - Choose k
    - Choose C; it may depend on your choice of k
    - Once you choose k and C, you must prove the truth of the implication (often by induction)

The Growth of functions (2.2) (cont.)

An alternative for those with a calculus background:

– **Definition:**

$$\textit{if } \lim_{n\to\infty} \frac{f(n)}{g(n)} = 0 \textit{ then } f \textit{ is } o(g) \textit{ (called little - o of } g)$$

– **Theorem:** If f is o(g) then f is O(g).

Proof: by definition of limit as n goes to infinity, f(n)/g(n) gets arbitrarily small.

That is for any $\varepsilon > 0$, there must be an integer N such that when $n > N$, $| f(n)/g(n) | < \varepsilon$.

Hence, choose $C = \varepsilon$ and $k = N$.

Q. E. D.

The Growth of functions (2.2) (cont.)

– It is usually easier to prove f is o(g)
- using the theory of limits
- using L'Hospital's rule
- using the properties of logarithms

etc.

– Example: 3n + 5 is $O(n^2)$

Proof: Using the theory of limits, it's easy to show

$$\lim_{n\to\infty} \frac{3n+5}{n^2} = 0$$

Hence 3n + 5 is $o(n^2)$ and so it is $O(n^2)$.

Q. E. D.

We will use induction later to prove the result from scratch.

The Growth of functions (2.2) (cont.)

- Also note that O(g) is a set called a

complexity class.

- It contains all the functions which g dominates.

f is O(g) means $f \in O(g)$.

The Growth of functions (2.2) (cont.)

- Properties of Big-O
  - f is O(g) iff $O(f) \subseteq O(g)$
  - If f is O(g) and g is O(f) then O(f) = O(g)
  - The set O(g) is closed under addition :
    If f is O(g) and h is O(g) then f + h is O(g)
  - The set O(g) is closed under multiplication by a scalar *a* (real number):
    If f is O(g) then a*f is O(g)
    that is,

    O(g) is a vector space.

    (The proof is in the book).
    Also, as you would expect,
  - if f is O(g) and g is O(h), then f is O(h).
    In particular

    $O(f) \subseteq O(g) \subseteq O(h)$

The Growth of functions (2.2) (cont.)

– **Theorem:**

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

- f_1f_2 is $O(g_1g_2)$ (1)
- $f_1 + f_2$ is $O(\max\{g_1, g_2\})$ (2)

The Growth of functions (2.2) (cont.)

– **Proof of (2):** There is a k_1 and C_1 such that

$f_1(n) < C_1g_1(n)$

when $n > k_1$.

There is a k_2 and C_2 such that

$f_2(n) < C_2g_2(n)$

when $n > k_2$.

We must find a k_3 and C_3 such that

$f_1(n)f_2(n) < C_3g_1(n)g_2(n)$

when $n > k_3$.

We use the inequality

$$\text{if } 0 < a < b \text{ and } 0 < c < d \text{ then } ac < bd$$

to conclude that

$$f_1(n)f_2(n) < C_1C_2g_1(n)g_2(n)$$

as long as $k > \max\{k_1, k_2\}$ so that both inequalities 1 and 2. hold at the same time.

Therefore, choose

$$C_3 = C_1C_2 \text{ and } k_3 = \max\{k_1, k_2\} \qquad \text{Q.E.D.}$$

The Growth of functions (2.2) (cont.)

- Important Complexity Classes

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

where $j>2$ and $c>1$.

The Growth of functions (2.2) (cont.)

– **Example:**

Find the complexity class of the function

$$(nn!+ 3^{n+2} + 3n^{100})(n^n + n2^n)$$

Solution:

This means to <u>simplify</u> the expression.

Throw out stuff which you know doesn't grow as fast.

We are using the property that if f is O(g) then f+g is O(g).

The Growth of functions (2.2) (cont.)

– Solution (cont.)

- Eliminate the $3n^{100}$ term since n! grows much faster.
- Eliminate the 3^{n+2} term since it also doesn't grow as fast as the n! term.
  Now simplify the second term:
  Which grows faster, the n^n or the $n2^n$?
- Take the log (base 2) of both.
  Since the log is an increasing function whatever conclusion we draw about the logs will also apply to the original functions (why?).
- Compare n log n or log n + n.
- n log n grows faster so we keep the n^n term
  The complexity class is
  $$O(n\ n!\ n^n)$$

The Growth of functions (2.2) (cont.)

– If a flop takes a nanosecond, how big can a problem be solved (the value of n) in

- a minute?
- a day?
- a year?

for the complexity class O(n n! nn).

– Note: We often want to compare algorithms in the same complexity class

The Growth of functions (2.2) (cont.)

– **Example:**

Suppose

Algorithm 1 has complexity $n^2 - n + 1$

Algorithm 2 has complexity $n^2/2 + 3n + 2$

Then both are $O(n^2)$ but Algorithm 2 has a smaller leading coefficient and will be faster for large problems.

Hence we write

Algorithm 1 has complexity $n^2 + O(n)$

Algorithm 2 has complexity $n^2/2 + O(n)$

Complexity of Algorithms (2.3)

- Time Complexity: Determine the approximate number of operations required to solve a problem of size n.
- Space Complexity: Determine the approximate memory required to solve a problem of size n.

Complexity of Algorithms (2.3) (cont.)

- Time Complexity
  - Use the Big-O notation
  - Ignore house keeping
  - Count the expensive operations only
  - Basic operations:
    - searching algorithms - key comparisons
    - sorting algorithms - list component comparisons
    - numerical algorithms - floating point ops. (flops) - multiplications/divisions and/or additions/subtractions

Complexity of Algorithms (2.3) (cont.)

- **Worst Case:** maximum number of operations
- **Average Case:** mean number of operations assuming an input probability distribution

Complexity of Algorithms (2.3) (cont.)

- Examples:
  - Multiply an n x n matrix A by a scalar c to produce the matrix B:

```
procedure (n, c, A, B)
    for i from 1 to n do
        for j from 1 to n do
                B(i, j) = cA(i, j)
        end do
    end do
```

Analysis (worst case):

Count the number of floating point multiplications.

n^2 elements requires n^2 multiplications.

time complexity is

$O(n^2)$ or *quadratic* complexity.

Complexity of Algorithms (2.3) (cont.)

- Multiply an n x n *upper triangular* matrix A

  $A(i, j) = 0$ if $i > j$

  by a scalar c to produce the (upper triangular) matrix B.

```
procedure (n, c, A, B)
/* A (and B) are upper triangular */
    for i from 1 to n do
        for j from i to n do
                B(i, j) = cA(i, j)
        end do
    end do
```

Analysis (worst case):

Count the number of floating point multiplications.

Complexity of Algorithms (2.3) (cont.)

The maximum number of non-zero elements in an n x n upper triangular matrix

$$= 1 + 2 + 3 + 4 + \ldots + n$$

or

- remove the diagonal elements (n) from the total (n^2)
- divide by 2
- add back the diagonal elements to get

$$(n^2 - n)/2 + n = n^2/2 + n/2$$

which is

$$n^2/2 + O(n).$$

Quadratic complexity but the leading coefficient is 1/2

Complexity of Algorithms (2.3) (cont.)

– Bubble sort: L is a list of elements to be sorted.

- We assume nothing about the initial order
- The list is in ascending order upon completion.

Analysis (worst case):
Count the number of list comparisons required.

Method: If the jth element of L is larger than the (j + 1)st, swap them.

Note: this is not an efficient implementation of the algorithm

Complexity of Algorithms (2.3) (cont.)

```
procedure bubble (n, L)
/*
   - L is a list of n elements
   - swap is an intermediate swap location
*/
   for i from n - 1 to 1 by -1 do
      for j from 1 to i do
         if L(j) > L(j + 1) do
               swap = L(j + 1)
               L(j + 1) = L (j)
               L(j) = swap
         end do
      end do
   end do
```

Complexity of Algorithms (2.3) (cont.)

- Bubble the largest element to the 'top' by starting at the bottom - swap elements until the largest in the top position.
- Bubble the second largest to the position below the top.
- Continue until the list is sorted.

n-1 comparison on the first pass

n-2 comparisons on the second pass

.

.

.

1 comparison on the last pass

Total:

$(n - 1)+ (n - 2) + \dots + 1 = O(n^2)$ or quadratic complexity

(what is the leading coefficient?)

Complexity of Algorithms (2.3) (cont.)

– An algorithm to determine if a function f from A to B is an injection:

Input: a table with two columns:

- Left column contains the elements of A.
- Right column contains the images of the elements in the left column.

Analysis (worst case):

Count comparisons of elements of B.

Recall that two elements of column 1 cannot have the same images in column 2.

Complexity of Algorithms (2.3) (cont.)

One solution:

- Sort the right column

  Worst case complexity (using Bubble sort)

$$O(n^2)$$

- Compare adjacent elements to see if they agree

  Worst case complexity

$$O(n)$$

Total:

$$O(n^2) + O(n) = O(n^2)$$

Can it be done in linear time?