Foundations of Discrete Mathematics

Chapters 8

By Dr. Dalia M. Gil, Ph.D.

Algorithms

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

Algorithms

- The word algorithm evolved from the older word algorism which is a corruption of the surname of a Persian mathematician Abu Ja’far Muhammad ibn Mûsâal-Khâwrismî

Example: Algorithms

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

a) First method: Use the English language to describe the sequence of steps.

b) Second method: Use a form of pseudocode.

Example Using English language

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.

Example Using English language

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

- Pseudocode provides an intermediate step between an English description of an algorithm and an implementation for this algorithm in a programming language.

- The steps of the algorithms are specified using instructions resembling those used in a programming language.

Example Using Pseudocode

```
function max($a_1, a_2, \ldots, a_n$: integers)
  max = $a_1$;
  for i=2 to n
        if max < $a_i$ then max = $a_i$;
```

← a_1 is assigned to max

← The loop examines all terms

↑ If a term is greater than the current value of max , it will be the new max

{Conclusion: max is the largest element}

Properties of Algorithms

- Input. An algorithm has input values from a specified set.
- Output. From each set of input values are the solution to the problem.
- Definiteness. The steps of an algorithm must be defined precisely.

Properties of Algorithms

- Finiteness. An algorithm should produce the desired output after a finite (but perhaps large) number of steps for any input in the set.
- Effectiveness. It must be possible to perform each step of an algorithm exactly and in a finite amount of time.

Properties of Algorithms

- Generality. The procedure should be applicable for all problems of the desired form, not just for a particular set of input values.

Properties of Algorithm 1

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

- The Input is a sequence of integers
- The Output is the largest integer in the sequence.

Properties of Algorithm 1

- Each step of this algorithm is precisely defined, since only assignments, a finite loop, and conditional statement occur.
- To show that the algorithm is correct when the algorithm terminates, the value of the variable max must be equal the maximum of the terms of the sequence.

Properties of Algorithm 1

- The algorithm uses a finite number of steps, since it terminates after all the integers in the sequence have been examined.
- The algorithm is general, since it can be used to find the maximum of any finite sequence of integers.

Example Algorithm 2

- Describe an algorithm whose input is a list $a_1, a_2, \ldots, a_n$ of integers and whose output is their sum.

1. Set S = 0;
2. for i=1 to n, replace S by S + a_i;   ← loop
3. Output S.

{The value of S output at Step 3 is the desired sum}

Algorithm translates into MATLAB

```
% Input a list a1, a2, ..., an of integers and
% whose output is their sum
a = [5 17 3 6 4];
n = 5;
s = 0;
for i=1:n
    s = s + a(i);
end
s
```

Algorithm translates into C

```
#include <stdio.h>
int main(void)
{
    int a[5]={5, 17, 3, 6, 4}, s, i;
    s = 0;
    for (i = 0; i < 5; i++)
        s = s + a[i];
    printf("s = %d\n", s);
    return 0;
}
```

Horner's Algorithm

- Given integers a_0, a_1, ..., a_n and an integer x, to evaluate the expression $a_0 + a_1x + a_1x^2 + \ldots + a_nx^n$,.

1. Set S = a_n;
2. for i=1 to n, replace S by a_{n-i} + Sx; ← loop
3. Output S.

{The value of S is the desired number $a_0 + a_1x + a_1x^2 + \ldots + a_nx^n$}

Example of Horner’s Algorithm

- Given $f(x) = -1 + 2x + 4x^2 - 3x^3$ and $x = 5$

1. Set $S = a_3 = -3$ $n = 3$

$$a_{n-i} + Sx$$

1. $i=1$: replace S by $a_2 + Sx = 4 – 3(5) = -11$
   $i=2$: replace S by $a_1 + Sx = 2 – 11(5) = -53$
   $i=3$: replace S by $a_0 + Sx = -1 – 53(5) = -266$.
   Since $i=n=3$, Step 2 is complete.
2. Output $S = -266$

Example with MATLAB

```
%Horner's Algorithm
% Evaluate f(x) = -1 + 2x + 4x^2 - 3x^3, x = 5
a = [-1 2 4 -3];
x = 5;
n = 4
s= a(n)
for i = 1:4
    if (n-i)== 0, break, end
  s = a(n-i) + s*x;
end
s
```

Example with C language

```
// Horner’s Method
#include <stdio.h>

int main (void)

{
     int a[] = { -1, 2, 4, -3}, x =5, i, s;

     s = a[3] = -3;
     for (i=1; i<=3; i++)
          s = a[3-i] + s*x  ;
    printf("S = %d\n", s);

     return 0;
}
```

Example Algorithm

- Describe an algorithm that, upon input of a list $a_1, a_2, \ldots, a_n$, output it.

```
1. output $a_1$;
2. if n=1, stop;
   else for i = 2 to n,
           if $a_i$ does not equal any of $a_1, a_2, \ldots, a_n$
```

The algorithms outputs the distinct items among $a_1, a_2, \ldots, a_n$

Searching Algorithms

- The problem of locating an element in an ordered list occurs in many context and are called searching problems.

The General Searching Problem

- Locate an element x in a list of distinct elements a_1, a_2, …,a_n, or determine that it is not in the list.

- The solution to this search problem is the location of the term in the list that equals x (that is, i is the solution if $x = a_i$) and is 0 if x is not in the list.

Some Searching Algorithms

- The linear search or sequential search algorithm.
- The binary search algorithm

The Linear Search Algorithm

- Begins by comparing x and a_1, when $x = a_1$, the solution is the location of a_1, namely 1.
- When $x \neq a_1$, compare a_1 with a_2. If $x = a_2$, the solution is the location of a_2, namely 2.

The Linear Search Algorithm

- When $x \neq a_2$, compare a_1 with a_3. Continue this process, comparing x successively with each term of the list until a match is found, where the solution is the location of that term.
- Unless no match occurs. If the entire list has been searched without locating x, the solution is 0.

The Linear Search Algorithm (1)

```
to search a list $a_1, a_2, \ldots, a_n$ for the element x
i = 1
while ($i \le n$ and $x \ne a1$)
    i = i + 1
if $i \le n$ then location = i
    else location = 0
{location is the subscript of the term that
  equals x, or is 0 if x is not found}
```

The Linear Search Algorithm (2)

```
to search a list $a_1, a_2, \ldots, a_n$ for the element x
for  i = 1 to n
    if x = $a_i$, output “true” and set i = 2n;
    if i ≠ 2n, output “false.”
```

Setting i = 2n is a little trick that stops the loop as soon as x has been found and, if x is not in the list, ensures that “false” is output at the end.

For $x = -2$ and a list $a_1 = 6$, $a_2 = 0$, $a_3 = -2$, and $a_4 = 1$

set $i = 1$, $x \neq a_1 = 6$,

set $i = 2$, $x \neq a_2 = 0$,

Since $i = a_3 = -2$, it outputs "true" and

Sets $i = 2n = 8$.

Since i is no longer in the range from 1 to n, the loops stops.

For $x = 2$ and a list $a_1 = 6$, $a_2 = 0$, $a_3 = -2$, and $a_4 = 1$

set $i = 1$, $x \neq a_1 = 6$,

set $i = 2$, $x \neq a_2 = 0$,

set $i = 3$, $x \neq a_3 = -2$,

set $i = 4$, $x \neq a_4 = 1$,

$i = 4 \neq 2n$, so the algorithm outputs “false.”

The Binary Search Algorithm

- Binary algorithm is used when the list has terms occurring in order of increasing size.

- If the terms are numbers, they are listed from smallest to largest; if they are words are listed lexicographic, or alphabetic order.

The Binary Search Algorithm

- It proceeds by comparing the element to be located to the middle term of the list.

- The list then is split into two smaller sublists of the same size, or where one of these smaller has one fewer term than the other.

The Binary Search Algorithm

- The search continue by restricting the search to the appropriate sublist based on the comparison of the element to be located and the middle term.

The Binary Search Algorithm

- To search for 19 in the list

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

- First split the list, which has 16 terms, into two smaller list with eight terms each, namely

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

The Binary Search Algorithm

- Then, compare 19 and the largest term in the first list. Since 10 < 19, the first list is disregarded.

- The second list 12 13 15 16 18 19 20 22 is split in two smaller lists of four terms each

12 13 15 16 18 19 20 22

The Binary Search Algorithm

- Since 16 < 19, the first list is disregarded, and the search is restricted to the second of these lists 18 19 20 22

- The, this list is split in two smaller lists of two terms each

18 19 20 22

The Binary Search Algorithm

- Since 19 is not greater than the largest term of the first of these two list, which is also 19,

- the search is restricted to the first list: 18, 19, which contains the 13th. And 14th. Terms of the original list.

The Binary Search Algorithm

- Then list is split in two lists of one term each: 18 and 19. Since 18 < 19, the search is restricted to the second list containing the 14th term of the original list, which is 19.

- Now the search has been narrowed down to one term, a comparison is made, and 19 is located as the 14th. term in the original list.

The Binary Search Algorithm

- Begins by comparing x with middle term of the sequence, a_m, where $m = \lfloor (n+1)/2 \rfloor$.
- Note: $\lfloor x \rfloor$ is the greatest integer not exceeding x.

The Binary Search Algorithm

- If $x > a_m$, The search for x can be restricted to the second half of the sequence, which is $a_{m+1}, a_{m+2}, \ldots, a_n$.
- If x is not greater than a_m, the search for x can be restricted to the first half of the sequence, which is $a_1, a_2, \ldots, a_m$.

The Binary Search Algorithm

- The search has now been restricted to a list with no more than $\lceil n/2 \rceil$ elements.
- Note: $\lceil x \rceil$ is the smallest integer term of the restricted list.

The Binary Search Algorithm

- Then restrict the search to the first or second half of the list.
- Repeat this process until the list with one term is obtained.
- Then determine whether this term is x.

The Linear Search Algorithm

- Begins by comparing x and a_1, when $x = a_1$, the solution is the location of a_1, namely 1.
- When $x \neq a_1$, compare a_1 with a_2. If $x = a_2$, the solution is the location of a_2, namely 2.

To search for an element x in an ordered list $a_1 \le a_2 \le \ldots \le a_n$ proceed as follows

```
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⌋
    i = i + 1
    if x > a_m then i = m + 1 else j = m
end
if x = a_i then location = i else location = 0
```

{location is the subscript of the term equals x}

To search for an element x in an ordered list $a_1 \le a_2 \le \ldots \le a_n$ proceed as follows

```
**while** $n > 0$
    **if** $n = 1$ then
        **if** $x = a_1$ output “true” and set $n = 0$;
        **else** output false and set $n = 1$;
    **else**
    set $m = \lfloor n/2 \rfloor$;
    **if** $x \le a_m$ replace the current list with $a_1, \ldots, a_m$ and set $n = m$;
    **else** replace the current list with $a_{m+1}, \ldots, a_n$ and replace n by $n - m$.
**end**
```

Sorting

- Ordering the elements of a list is a problem that occurs in many contexts.
- A sorting is putting the elements into a list in which the elements are in creasing order.

Sorting

- Sorting the list 7, 2, 1, 4, 5, 9 produces the list 1, 2, 4, 5, 7, 9
- Sorting the list d, h, c, a, f (using alphabetic order) produces the list a, c, d, f, h.

Sorting

- Some algorithms are easier to implement.
- Some algorithms are more efficient.
- Some algorithms take advantages of particular computer architectures.
- Some algorithms are particular clever.

Some Sorting Algorithms

- The bubble sort algorithm.
- The insertion sort algorithm.

The Bubble Sort Algorithm

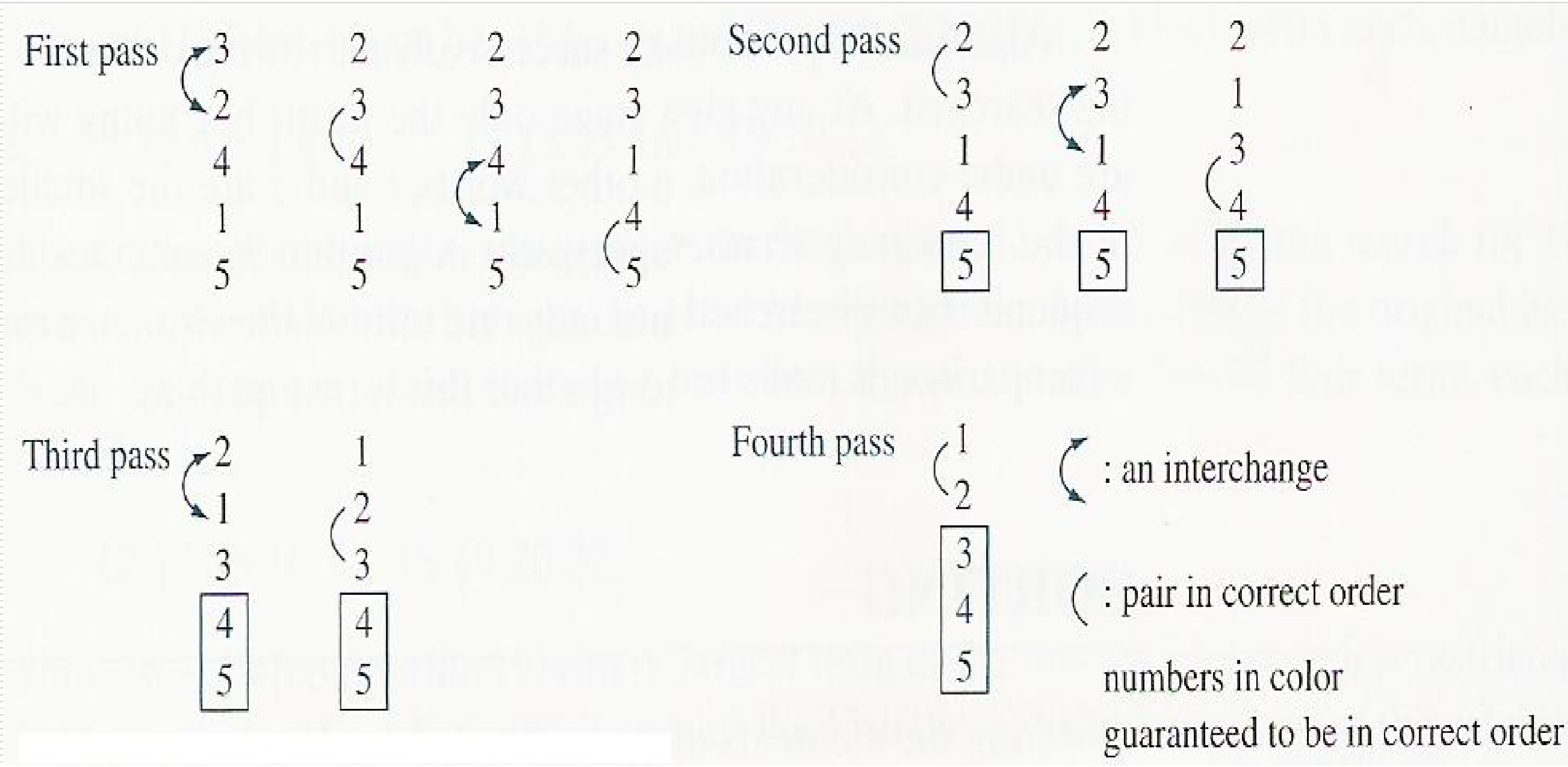
- The bubble sort algorithm is one of the simplest sorting algorithms, but not one of the most efficient.

- It put a list into increasing order by successively comparing adjacent elements, interchanging them if they are in the wrong order.

The Bubble Sort Algorithm

- To carry out the bubble sort algorithm, the basic operation is interchange a large element with a smaller one following it, starting at the beginning of the list., for a full pass.
- This process is repeated until the sort is complete.

The Bubble Sort Algorithm



“Discrete Mathematics and its Applications.” Fifth Edition, by Kenneth H. Rosen. Mc Graw Hill, 2003. pag 126

The Bubble Sort Algorithm

- http://www.cs.bme.hu/~gsala/alg_anims/3/bsort-e.html

The Bubble Sort Algorithm

```
**function** bubblesort ($a_1, a_2, \ldots, a_n$)
  **for** i = 1 **to** n – 1
    **for** j = 1 **to** n – i
      **if** $a_j > a_{j+1}$ **then** interchange $a_j$ and $a_{j+1}$

{$a_1, a_2, \ldots, a_n$ is increasing order}
```

The Bubble Sort Algorithm

To sort n elements $a_1, a_2, \ldots, a_n$ from least to greatest

```
for i = n – 1 down to 1
    for j = 1 to i
        if $a_j > a_{j+1}$ swap $a_j$ and $a_{j+1}$

{$a_1, a_2, \ldots, a_n$ is increasing order}
```

The Insertion Sort Algorithm

- The insertion sort is a simple sorting algorithm, but it is usually not the most efficient.
- To sort a list with n elements, the insertion sort begins with the second element.

The Insertion Sort Algorithm

- The insertion sort compares this second element with the first element and insert if before the first element.
- If it does not exceed the first element and after the first element if it exceeds the first element.

The Insertion Sort Algorithm

- The third element is then compared with the first element, and if it is larger than the first element,
- It is compared with the second element; it is inserted into the correct position among the first three elements.

The Insertion Sort Algorithm

- In general, in the jth step of the insertion sort, the jth element of the list is inserted into the correct position in the list of the previously sorted j – 1 elements.

- To insert the jth element in the list, a linear search technique is used;

The Insertion Sort Algorithm

- the jth element is successively compared with the already sorted j – 1 elements at the start of the list until the first element that is not less than this element is found

- or until it has been compared with all j – 1 elements; the jth element is inserted in the correct position so that the first j elements are sorted.

The Insertion Sort Algorithm

- Use the insertion sort to put the elements of the list 3, 2, 4, 1, 5 in increasing order.

1. Compare 2 and 3. Since 3 > 2, it places 2 in the first position. Now 2, 3, 4, 1, 5.
2. The third element 4 is inserted and compared with 2 (4 > 2) and 3 (4 > 3). 4 is placed in the third position. Now the list is 2, 3, 4, 1, 5

The Insertion Sort Algorithm

3. Next, find the correct place for the fourth element, 1, among the already sorted elements 2, 3, 4.
4. Since 1 < 2, we obtain the list 1, 2, 3, 4, 5
5. Finally, we insert 5 in to the correct position by successively comparing it to 1, 2, 3, and 4.
6. Since 5 > 4, it goes at the end of the list, producing the correct order.

function insertion sort ($a_1, a_2, \ldots, a_n$: real numbers with $n \geq 2$)

```
for j = 2 to n
begin
    i = i
    while $a_j > a_i$
            i = i + 1
    m = $a_j$
    for k = 0 to j - i – 1
        $a_{j-k} = a_{j-k-1}$
    $a_j$ = m
end {$a_1, a_2, \ldots, a_n$ are sorted}
```

Merging Algorithm

- To merge two given sorted lists L_1: $a_1 \leq a_2 \leq \ldots \leq a_s$, and L_2: $b_1 \leq b_2 \leq \ldots \leq b_t$ of lengths s and t, into a single sorted list L_3: $c_1 \leq c_2 \leq \ldots \leq c_{s+t}$ of length s + t, proceed as follows:

Step 1: Set L_3 equal to an empty list

Step 2: If L_1 is empty, set $L_3 = L_2$ and stop. If L_2 is empty, set $L_3 = L_1$ and stop.

Merging Algorithm

Step 3: Suppose $a_1 \leq b_1$ then remove a_1 from L_1 and append it to L_3 ; if this empties L_1, append the elements of L_2 to L_3 and stop.

If $r > 0$ elements remain in L_1, label them $a_1, a_2, \ldots , a_r$ in increasing order and repeat Step 3.

Merging Algorithm

Step 3 (cont.): Suppose $a_1 > b_1$ then remove b_1 from L_2 and append it to L_3 ; if this empties L_2, append the elements of L_1 to L_3 and stop.

If $r > 0$ elements remain in L_2, label them $b_1, b_2, \ldots, b_r$ in increasing order and repeat Step 3.

Example: Merging Algorithm

- Apply the Merging algorithm to the lists L_1: a_1 a_2 a_3, and L_2: b_1 b_2 b_3

  3 5 8 1 7 8
- The lists are not empty.

Step 3: $a_1 = 3 > b_1 = 1$

Append b_1 to the list L_3, which was initially empty.

Example: Merging Algorithm

Step 3 (cont.):

Relabel the remaining elements 7 and 8 of L_2 as b_1, b_2, respectively.

The lists are

L_1: a_1	a_2	a_3,	L_2: b_1	b_2	and	L_3: c_1
3	5	8	7	8		1

Example: Merging Algorithm

- Step 3 (cont.) : $a_1 = 3 \le b_1 = 7$

  Append a_1 to the list L_3, and relabel the remaining elements 5 and 8 of L_1 as a_1, a_2, respectively. The lists are

L_1:	a_1	a_2,	L_2:	b_1	b_2	and	L_3:	c_1	c_2
	5	8		7	8			1	3

Example: Merging Algorithm

- **Step 3 (cont.)** : $a_1 = 5 \le b_1 = 7$

  Append a_1 to the list L_3, and relabel the remaining element 8 of L_1 as a_1. The lists are

L_1:	a_1,	L_2:	b_1	b_2	and	L_3:	c_1	c_2	c_3
	8		7	8			1	3	5

Example: Merging Algorithm

- Step 3 (cont.) : $a_1 = 8 > b_1 = 7$

  Append b_1 to the list L_3, and relabel the remaining element 8 of L_2 as b_1. The lists are

L_1: a_1,	L_2: b_2	and	L_3: c_1	c_2	c_3	c_4
8	8		1	3	5	7

Example: Merging Algorithm

- Step 3 (cont.) : Since $a_1 = 8 = b_1 = 8$

  Append a_1 to the list L_3, giving

L_1:	L_2:	b_1	and	L_3: c_1	c_2	c_3	c_4	c_5
		8		1	3	5	7	8

Since L_1 is empty, append 8 (from L_2) to L_3

L_3:	c_1	c_2	c_3	c_4	c_5	c_6
	1	3	5	7	8	8

Merge Sort Algorithm

- To sort a list a_1 , a_2 , … , a_n into increasing order proceed as follows:

Step 1: Set F = 0 ← **F is a flag to stop the algorithm**

Step 2: for i=1 to n, let the list L_i be the single element a_i.

Merge Sort Algorithm

```
Step 3: While F = 0   ← F is a flag to stop the algorithm
    if n = 1, set F = 1 and output $L_1$;
    if n = 2m is even
        for i=1 to m
            * merge the sorted list $L_{2i-1}$ and $L_{2i}$
              and label the resulting sorted list $L_i$;
    set n = m.
```

Merge Sort Algorithm

```
Step 3 (cont.):
    if $n = 2m + 1 > 1$ is odd
        for i=1 to m
            * merge the sorted list $L_{2i-1}$ and $L_{2i}$
            and label the resulting sorted list $L_i$;
            * set $L_{m+i}$ and $L_i$
    set $n = m + 1$.
end while
```

Example: Merge Sort Algorithm

- Sort the following list

a_1	a_2	a_3	a_4	a_5	a_6	a_7
2	9	1	4	6	5	3

Step 1: Set F = 0

Step 2: list L_1, L_2, L_3 , L_4, L_5, L_6, L_7 are defined, each of length 1.

L_1:2 L_2:9 L_3 :1 L_4:4 L_5:6 L_6:5 L_7:3

Example: Merge Sort Algorithm

Step 3 :

$n = 2m + 1 > 1$ is odd $n = 2(3) + 1$, $m = 3$

form 4 new lists L_1, L_2, L_3 and L_4,

by merging the first six former lists in pairs into three and adding the seventh

L_1:2, 9 L_2:1, 4 L_3 : 6, 5 L_4:3

$n = m + 1 = 3 + 1 + 4$

Example: Merge Sort Algorithm

Step 3 (cont.) :

$n = 2m$ is even ($n = 2(2)$), $m = 2$

form 2 new lists L_1 and L_2,

by merging the 4 former lists L_1, L_2 and L_2, L_4, respectively.

L_1:1, 2 , 4, 9 L_2:3, 5, 6

Now n is replaced by $m = 2$

Example: Merge Sort Algorithm

Step 3 (cont.) :

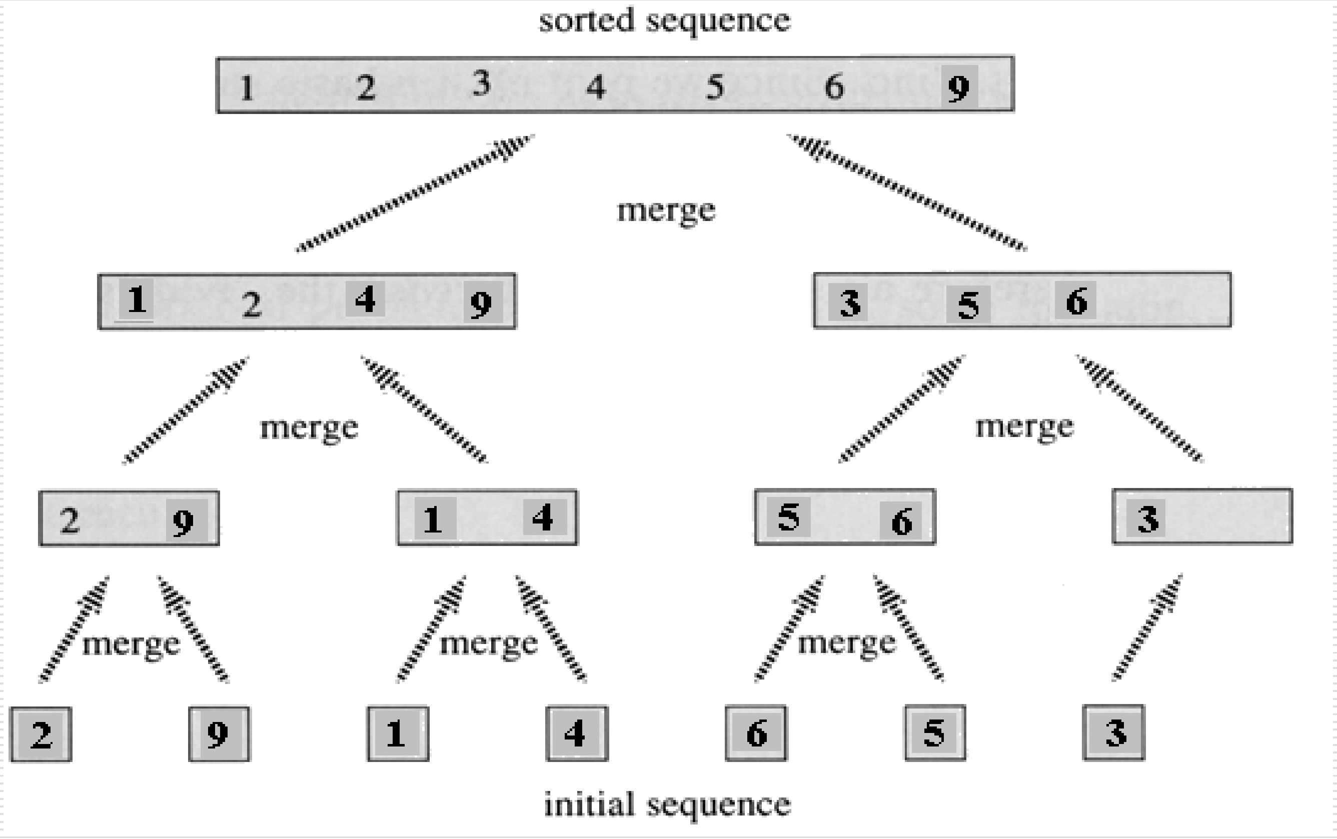
n = 2m is even (n = 2(1)), m = 1

form 1 new list by merging the 2 former lists L_1 and L_2, respectively.

L_1:1, 2 ,3, 4 , 5, 6, 9

Now n is replaced by m = 1. Since n = 1, set F = 1, output L_1, and stop.

Example: Merge Sort Algorithm



Example: Merge Sort Algorithm

- http://www.geocities.com/SiliconValley/Program/2864/File/Merge1/mergesort.html

Web with Sorting Algorithms

- http://www.cs.ubc.ca/~harrison/Java/sorting-demo.html

Complexity

- There are different measures of the efficiency of algorithms such as time, operation counts, amount of space to hold numbers in memory, and others.

Example: Complexity

- Find the complexity function for adding two n-digit integers if the basic operation is addition of single-digit integers.

- Suppose the integers to be added are $a = (a_{n-1}\ \ a_{n-2}\ \dots\ a_1\ a_0)_{10}$ and $b = (b_{n-1}\ \ b_{n-2}\ \dots\ b_1\ b_0)_{10}$ expressed in base 10 .

Example: Complexity

- $a_0 \leftarrow$ the units digit of a
- $b_0 \leftarrow$ the units digit of b
- $a_1 \leftarrow$ the tens digits of a
- $b_1 \leftarrow$ the tens digit of b

Example: Complexity

- The units digits a + b is obtained by adding a_0 and b_0 (a single operation).
- To obtain the tens digit, we add a_1 and b_1; then perhaps, we add 1, depending on whether there is a carry from the previous step.

Example: Complexity

- At most two single-digit additions (two operations) are required for the tens digits of a + b.
- Similarly, at most two operations are required for each digit of a + b after the units digit.
- An upper bound for the number operations is $f(n) = 1 + 2(n – 1) = 2n - 1$

Complexity

- Most complexity problems <u>is difficult to obtain an exact count for the number of single-digit additions required</u>.
- Complexity is measured in worst-case terms.
- The addition of two n-digit numbers requires at most 2n – 1 single-digit additions.

Complexity

- Complexity based on the number of operations that will never be exceeded. That is the upper-bound or worst-case.

Complexity

- http://www.geocities.com/SiliconValley/Network/1854/Sort1.html

Complexity

- The time required to solve a problem depend on:

1. the number of operations it uses.
2. The hardware used to run the program that implements the algorithm.

Complexity

- If the hardware and software change, the time required to solve a problem of size n can be approximated by multiplying the previous time required by a constant.
- On a supercomputer to solve a problem of size n a million times faster than on a PC.

Complexity: Big-Oh Notation

- Big-O notation estimates the growth of a function without worrying about constant multipliers or smaller order terms.

- Big-O notation does not consider the hardware or software used to implement the algorithm.

Complexity: Big-Oh Notation

- We can assume that the different operations used in an algorithm take the same time, which simplifies the analysis considerably.

- Big-O notation is used to estimate the number of operations an algorithm uses as its input grows.

Complexity: Big-Oh Notation

- We can determine whether it is practical to use a particular algorithm to solve a problem as the size of the input increases.

- We can compare two algorithms to determine which is more efficient as the size of the input grows.

Complexity: Big-Oh Notation

- Two algorithms for solving a problem, one using $100n^2 + 17n + 4$ operations and the other using n^3 operations,
- Big-O notation can help us see that the first algorithm uses far fewer operations when n is large, even though it uses more operations for small values of n, such as $n = 10$.

Complexity: Big-Oh Notation

- Let f and g be functions $N \rightarrow R$, f is Big Oh of g and is written $f = O(g)$ or $O(g(x))$ if there is an integer n_0 and a positive real number c such that

  $|f(n)| \leq c|g(n)|$ for all $n \geq n_0$.

  This is read as "f(x) is big-oh of g(x)."

Complexity: Big-Oh Notation

We can say

“There exists an integer n_0 such that $|f(n)| \le c|g(n)|$ for all $n \ge n_0$.”

“There exists an integer n_0 such that $|f(n)| \le c|g(n)|$ for all sufficiently large n.”

Complexity: Big-Oh Notation

- Instead of saying "There exists an integer n_0 such that $|f(n)| \leq c|g(n)|$ for all $n \geq n_0$."
- Say "$|f(n)| \leq c|g(n)|$ for all sufficiently large n."
- If f, g: $N \rightarrow R$. are functions that count operations, f(n) and g(n) are positive for all sufficiently large n, then the absolute value symbol around them <u>are not necessary.</u>

Complexity: Big-Oh Notation

- Let $f(n) = 15n^3$ and $g(n) = n^3$

$$|f(n)| \le c|g(n)| \text{ for all } n \ge n_0.$$

- With $n_0 = 1$ and $c = 15$, so $f = O(g)$

<u>f is Big Oh of g</u>

Example: Big-Oh Notation

- Show that $f(n) = n + 1$ and $g(n) = n^2$

  If $n \geq 1$, $f(n) \leq n + n = 2n$

because

$2n \leq 2n^2$

Taking $n_0 = 1$ $c = 2$ ← witnesses, $f = O(g)$

Complexity: Big-Oh Notation

- Show that $7x^2$ is $O(x^3)$

When $x > 7$, we have $7x^2 < x^3$

The inequality is obtained by multiplying both sides of $x > 7$ by x^2

$C = 1$ and $k = 7 \leftarrow$ witnesses

When $x > 1$, $7x^2 < 7x^3$, so that $C= 1$ and $k = 1$

are also witnesses to the relationship

$$7x^2 \text{ is } O(x^3)$$

Properties: Big-Oh Notation

- Let f, g, f_1, g_1 be functions $N \rightarrow R$

a) **If $f = O(g)$, then $f + g = O(g)$**

b) **If $f = O(f_1)$ and $g = O(g_1)$,**
then $f \cdot g = O(f_1 \cdot g_1)$

Complexity: Some Definitions

- If f and g are functions $N \to R$, we say that <u>f has smaller order than g</u> and write $f \prec g$ if and only if $f = O(g)$, but $g \neq O(f)$.

- If $f = O(g)$ and $g = O(f)$, then we say that f and g have <u>the same order</u> and write $f \asymp g$.

Complexity: Some Definitions

- $n + 1 \prec n^2$; thus $n + 1$ <u>has smaller order than</u> n^2

- $15n^3 \asymp n^3$: $15n^3$ and n^3 have <u>the same order</u>

Complexity: Some Propositions

- Let f, g be functions $N \rightarrow R$

  a) If $\lim_{n \to \infty} f(n)/g(n) = 0$, then $f \prec g$

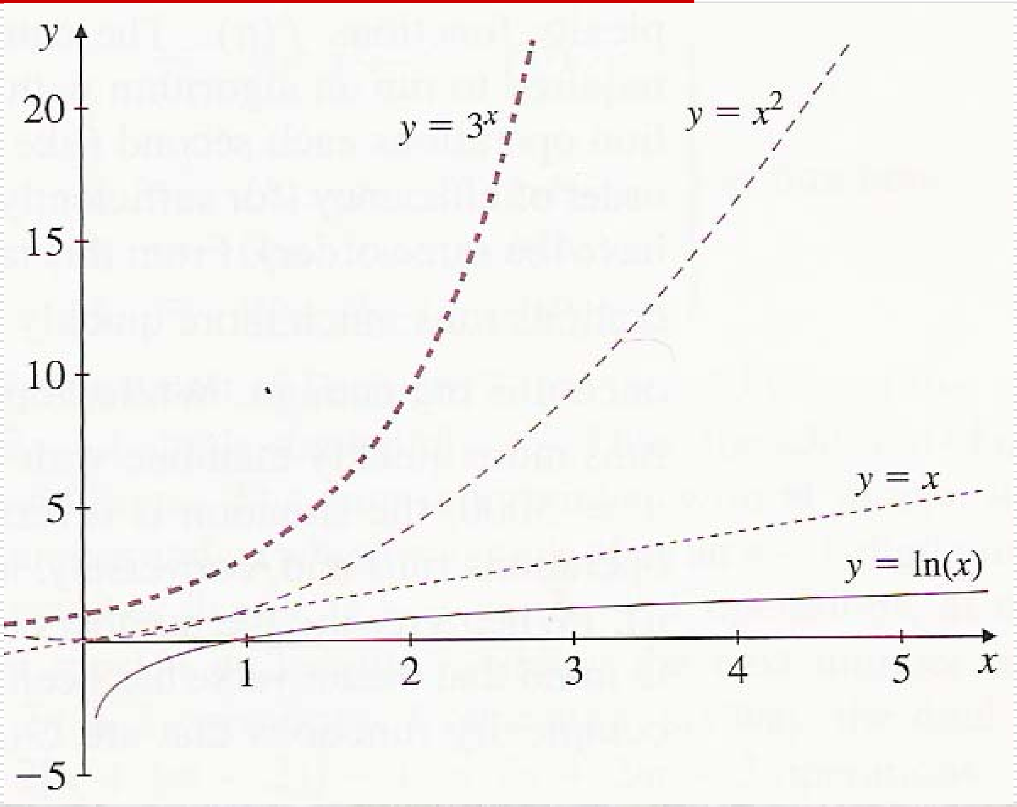
  b) If $\lim_{n \to \infty} f(n)/g(n) = \infty$, then $f \prec g$

  c) If $\lim_{n \to \infty} f(n)/g(n) =$ L for some number $L \neq 0$, then $f \asymp g$

Complexity: Some Propositions

- Suppose a and b are real numbers $a < b$. Then $n^a \prec n^b$
- $\log_b n \prec n$ for any real number b, $b > 1$
- $\log_a n \asymp \log_b n$ (two algorithm functions with bases larger than 1 have the same order.

Some Common Complexity Functions



$$1 \prec \log n \prec n \prec n^a \prec b^n \prec n! \prec n^n.$$

"Discrete Mathematics With Graph Theory." Third Edition, by E. G. Goodaire and M. M. Parmenter. Pearson Prentice Hall, 2006. pag 259

Topics covered

- Algorithms.
- Searching and Sorting.
- Complexity and Big-Oh notation

Reference

- "Discrete Mathematics with Graph Theory", Third Edition, E. Goodaire and Michael Parmenter, Pearson Prentice Hall, 2006. pp 247-280.

Reference

- “Discrete Mathematics and Its Applications”, Fifth Edition, Kenneth H. Rosen, McGraw-Hill, 2003. pp 120-152.