Enrichment Mini-Course Notes:

Inside Computer Programming

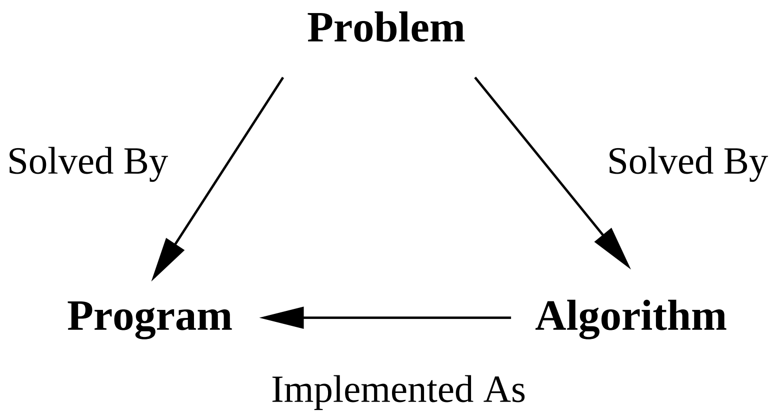
Todd Wareham

Department of Computer Science
Memorial University of Newfoundland
`harold@cs.mun.ca`

May 3–4, 2004

Table of Contents

Problems, Algorithms, and Programs



- A **problem** is a set of inputs with a set of associated outputs. Each input has one associated output.
- An **algorithm** is a sequence of instructions that solves a given problem, *i.e.*, given an input, the algorithm computes the corresponding output.
- A **program** is a sequence of instructions *in some computer language, e.g., FORTRAN, C, Java*, that solves a given problem.
- A problem may have many algorithms; an algorithm may be implemented as many programs; each program implements one algorithm.

Given a problem, which algorithm for that problem should we choose to implement in a program?

Problems

- **Example Problem #1**:

  "Given the radius of a circle, what is that circle's area?"

  $$\Downarrow$$

  Input: A radius r.
  Output: The area of a circle with radius r.

- **Example Problem #2**:

  "If I know the grade a student is in, what type of school are they in?"

  $$\Downarrow$$

  Input: A student grade-number g.
  Output: The type of school with students of grade-number g.

- **Example Problem #3**:

  "I have a list of numbers and I'd like to know what they add up to."

  $$\Downarrow$$

  Input: A list L of numbers.
  Output: The sum of all numbers in L

Algorithms: Basic Parts

- **Algorithm = data + instructions**
- The most basic types of data are single-value pieces of data (**variables**) or multiple-value indexed lists of data (**lists**).
  - Each variable or list has a name.
  - The ith element of list L has the name $L[i]$.
- There are three basic types of instructions:
  1. **Assignment**
     ```
     something = something else
     ```
  2. **Conditional** (if-then-else)
     ```
     IF (something is true) THEN
         do this
     ELSE
         do that
     ```
  3. **Loop**
     (a) **Conditional Loop**
     ```
     WHILE (something is true) DO
         this
     ```
     (b) **Counted Loop**
     ```
     FOR some number of times DO
         this
     ```

Algorithms:
Finding the Area of a Circle

Intuition:

"I know that the area of a circle with radius r is given by the expression $\pi \times r^2$, where $\pi = 3.14159$."

Algorithm:

```
area = 3.14159 * r * r
return(area)
```

Algorithms: Determining your School Type

Intuition:

"Let's see . . . If someone is in Grades 1 to 6, they are in elementary school. If they are in grades 7 to 9, they are in junior high school. Otherwise, they are in high school."

Algorithm:

```
if ((g >= 1) and (g <= 6)) then
    school_type = "elementary"
else if ((g >= 7) and (g <= 9)) then
    school_type = "junior high"
else if ((g >= 10) and (g <= 12)) then
    school_type = "high"
else
    school_type = "invalid"
return(school_type)
```

Algorithms:
Summing a List of Numbers
(Number of List Elements Specified)

Intuition:

"How about I start with some sum-variable set to 0, and I go through the list and add each list-element to the sum-variable?"

Algorithm:

```
sum = 0
for i = 1 to n do
    sum = sum + L[i]
return(sum)
```

Algorithms: Summing a List of Numbers (End of List Specified by -1-Value)

Intuition:

"Ohhhkaaaayyyy ... Once again, I start with some variable set to 0. However, this time, I'll go through the list and add each element to the sum-variable *until* I hit an element with value -1. This means I'll have to check each element before I add it to the sum-variable."

Algorithm:

```
sum = 0
i = 1
while (L[i] != -1) do
    sum = sum + L[i]
    i = i + 1
return(sum)
```

Algorithms: Finding the Smallest Element in a List

Problem:

Input: A list L with n elements.
Output: The position of the smallest element in L.

Intuition:

"If I don't know anything else about the list except that it has n elements, I suppose I'll have to look at all of the list-elements and keep track of the *position* of the smallest one I've seen so far (I can use that to get the value whenever I need it). When I've looked at the whole list, the smallest element I've found so far is the smallest element in the list."

Algorithm:

```
min_pos = 1
for i = 2 to n do
    if (L[i] < L[min_pos]) then
        min_pos = i
return(min_pos)
```

The Searching Problem

- The problem of finding if some value is included in a given list occurs in lots of applications, *e.g.*,
  - looking up a person's telephone number
  - charging some amount to a person's credit card
  - finding out if anyone won the Lotto 6-49 jackpot this week

LIST SEARCH
Input: A (unsorted/sorted) list L with n elements, a target-value t.
Output: The position of the element with value t in L if such an element exists and -1 otherwise.

- We will assume both list-elements and t are numbers for the sake of simplicity; however, our algorithms will work for any values that can be ordered, *e.g.*, character strings.

Unsorted-List Search Algorithms: Linear Search

Intuition:

"Well, if I don't know anything else about the list except that it has n elements, I suppose I'll have to look at each element in the list and see if it is equal to the target-value. If I find such an element, I can stop and return that element's position; otherwise, I return -1 after I've look at all elements in the list. Sounds like a lot of work. Bummer."

Algorithm:

```
t_pos = -1
i = 1
while ((i <= n) and (t_pos == -1)) do
    if (L[i] == t) then
        t_pos = i
    i = i + 1
return(t_pos)
```

Sorted-List Search Algorithms: Binary Search

Intuition:

"Hmmm ... Whenever I look at $L[i]$ where i is the middle of the list and $L[i]$'s not equal to the target-value, as L is sorted, I know that the target-value must be either above or below i in the list (depending on whether the target-value is greater or less than $L[i]$). I can keep repeating this in a loop until I either find the target-value or run out of list to search. Cool!"

Algorithm (Version #1):

```
set the current list to L
while we haven't found t in the list
      and there's still a current list
       to search do
    if t isin't the middle element of
        the current list then
        if t > middle element then
            set current list to upper
             part of current list
        else
            set current list to lower
             part of current list
```

Sorted-List Search Algorithms: Binary Search (Cont'd)

Algorithm (Version #2):

```
t_pos = -1
left = 1
right = n
while ((t_pos == -1) and
       (left <= right)) do
    t_pos = (left + right) / 2
    if (L[t_pos] != t) then
        if (t > L[t_pos]) then
            left = t_pos + 1
        else
            right = t_pos - 1
        t_pos = -1
return(t_pos)
```

Time Complexity: What Is It?

- There are typically many algorithms for solving a particular problem. Which one do we use?
  - ⇒ Use the simplest algorithm, *i.e.*, the algorithm that is easiest to understand and hence most likely to be implemented correctly in a program.
  - ⇒ Use the most **efficient** algorithm, *i.e.*, the algorithm that requires the least amount of computer time to solve the problem!
- As inputs may vary in size and an algorithm may require different amounts of time to compute on different inputs, measures of algorithm efficiency should be phrased relative input size.
- The **(worst-case) time complexity** of an algorithm is a function $O(f(n))$ of input size n that upper-bounds the amount of time required by that algorithm to solve its associated problem relative to an input of size n.
- By making appropriate abstractions, can make time complexity describe the running time of an algorithm *relative to any possible computer*.

Search Algorithms: Time Complexity

- You can get a rough estimate of worst-case time complexity by multiplying out the number of times each loop in the algorithm will execute in the worst case.
- For search algorithms, let the input size be n, the number of elements in the given list:
  - **Linear Search**: $O(n)$ time
  - **Binary Search**: $O(\log_2 n)$ time
- Where did this function $\log_2 n$ come from?
  - $\log_2 n$ is the logarithm (base 2) of n.
  - $\log_2 n$ is essentially the number of times you can divide n by 2 until you get a result that is less than or equal to one, *e.g.*, $log_2 4 = 2$, $log_2 7 \approx 3$, $log_2 16 = 4$.
  - As binary search discards half of the given list each time we iterate the main loop, this loop can iterate at most $\log_2 n$ times!

Search Algorithms: Time Complexity (Cont'd)

- Suppose you have a computer that executes a million instructions per second:

Input Size (n)	Time Complexity	
	B-Search ($\log_2 n$)	**L-Search** (n)
10	.000003 second	.00001 second
20	.000004 second	.00002 second
30	.000005 second	.00003 second
50	.000006 second	.00005 second
100	.000007 second	.0001 second
1000	.000010 second	.001 second
10000	.000013 second	.01 second
one million	.000020 second	1 second
300 million	.000028 second	5 minutes
five billion	.000032 second	1.38 hours

- Binary search much faster than linear search!
- Are there efficient algorithms for sorting lists?

The Sorting Problem

- The problem of sorting a given list occurs whenever you want to set a list up for fast search. It can also have other uses, *e.g.*, detecting duplicate values.

> LIST SORT
> **Input**: A list L with n elements.
> **Output**: The sorted version of L.

- As with the list search problem, we will assume list-elements are numbers for the sake of simplicity; however, our algorithms will work for any values that can be ordered, *e.g.*, character strings.

Sorting Algorithms: Selection Sort

Intuition:

"The first element in a sorted list is the smallest in the list, the second element is the smallest among the remaining elements in the list, and so on. Perhaps we could use our find-list-minimum algorithm in a loop!"

Algorithm (Version #1):

```
for i = 1 to n - 1 do
    find minimum element in L[i .. n]
    swap minimum element and element i
```

Algorithm (Version #2):

```
for i = 1 to n - 1 do
    min_pos = i
    for scan = i + 1 to n do
        if (L[scan] < L[min_pos]) then
            min_pos = scan
    temp = L[min_pos]
    L[min_pos] = L[i]
    L[i] = temp
```

Sorting Algorithms: Insertion Sort

Intuition:

"Suppose the first $i-1$ elements of L are sorted already; to add $L[i]$ to this sorted list, all I need to do is shift elements of $L[1..(i-1)]$ upwards until the spot where $L[i]$ fits is open. Say, I can loop this operation as well to add each element in $L[2..n]$ successively to the trivially sorted list $L[1..1]$! Sweet!"

Algorithm (Version #1):

```
for i = 2 to n do
    shift elements of L[1 .. (i - 1)]
     upwards until L[i]'s spot is open
    copy L[i] into open spot
```

Algorithm (Version #2):

```
for i = 2 to n do
    val = L[i]
    new_pos = i
    while ((new_pos > 1) and
           (L[new_pos - 1] > val)) do
        L[new_pos] = L[new_pos - 1]
        new_pos = new_pos - 1
    L[new_pos] = val
```

Sorting Algorithms: Bubble Sort

Intuition:

"In order for a list to be unsorted, there must be at least one pair of adjacent list-elements $L[i]$ and $L[i+1]$ such that $L[i] > L[i+1]$. Suppose we kept going through the list, swapping bad adjacent list-element pairs until there weren't any more such pairs?"

Algorithm (Version #1):

```
while list is not sorted do
    traverse L, swapping bad adjacent
     list-element pairs as necessary
    if no swaps occurred then
        the list is sorted
```

Sorting Algorithms: Bubble Sort (Cont'd)

Algorithm (Version #2):

```
sorted = false
while (not sorted) do
     num_swap = 0
     for i = 2 to n do
          if (L[i - 1] > L[i]) then
               temp = L[i - 1]
               L[i - 1] = L[i]
               L[i] = temp
               num_swap = num_swap + 1
     if (num_swap == 0) then sorted = true
```

Algorithm (Version #3):

```
sorted = false
while (not sorted) do
     sorted = true
     for i = 2 to n do
          if (L[i - 1] > L[i]) then
               temp = L[i - 1]
               L[i - 1] = L[i]
               L[i] = temp
               sorted = false
```

Sorting Algorithms: Time Complexity

Selection Sort:

- Outer loop always executes $n-1$ times.
- Inner loop executes $O(n)$ times, regardless of how sorted the input list is $\Rightarrow O(n^2)$ time.
- In practice, has lowest worst-case running time.

Insertion Sort:

- Outer loop always executes $n-1$ times.
- In the worst case (list sorted in reverse order), the inner loop executes $O(n)$ times $\Rightarrow O(n^2)$ time.
- In practice, has lowest average-case running time.

Bubble Sort:

- Inner loop always executes $n-1$ times.
- In the worst case (list sorted in reverse order), the outer loop executes $O(n)$ times $\Rightarrow O(n^2)$ time.
- In practice, has lowest best-case running time.

Sorting Algorithms: Time Complexity (Cont'd)

- Once again, on our megaflop computer . . .

Input Size (n)	Time Complexity		
	B-Search ($\log_2 n$)	L-Search (n)	**S/I/B-Sort** (n^2)
10	.000003 second	.00001 second	.0001 second
20	.000004 second	.00002 second	.0004 second
30	.000005 second	.00003 second	.0009 second
50	.000006 second	.00005 second	.0025 second
100	.000007 second	.0001 second	.01 second
1000	.000010 second	.001 second	1 second
10000	.000013 second	.01 second	1.67 minutes
one million	.000020 second	1 second	11.57 days
300 million	.000028 second	5 minutes	3 centuries
five billion	.000032 second	1.38 hours	7927 centuries

- The worst-case running time won't occur that often, so this isin't as bad as it seems. That being said, is there a sorting algorithm with a better worst-case running time? How about for restricted cases of the sorting problem?

Sorting Algorithms: Merge Sort

Intuition:

"You can sort a list L by separately sorting the top and bottom halves of the list and then merging these lists together to create a full sorted list. How do you sort each of the half-lists? Do the same thing on each half-list! You can keep splitting lists in this fashion until you have lists of length one, which are trivially sorted. Then you can backtrack and do the merging. Sounds bizarre, but it should work ..."

Algorithm (Version #1):

```
MergeSort(L,ls,lf)
    if there is more than one element
           in the current sublist then
        sort the bottom half of list L
        sort the top half of list L
        merge the top and bottom half
         sorted sublists
```

where `ls` (`lf`) is the start (final) index of the current sublist being sorted.

Sorting Algorithms: Merge Sort

Algorithm (Version #2):

```
MergeSort(L, ls, lf)
    if (ls < lf) then
        lm = FLOOR((ls + lr) / 2)
        MergeSort(L,ls,lm)
        MergeSort(L,lm,lf)
        Merge(L,ls,lm,lf)

Merge(L,ls,lm,lf)
    nb = (lm - ls) + 1
    nt = (lf - lm) + 1
    Create arrays BH and TH
    for i = 1 to nb do
        BH[i] = L[ls + (i - 1)]
    for i = 1 to nt do
        TH[i] = L[lm + (i - 1)]
    BH[nb + 1] = INFINITY
    TH[nt + 1] = INFINITY
    i = j = 1
    for k = ls to lf do
        if (BH[i] <= TH[j]) then
            L[k] = BH[i]
            i = i + 1
        else
            L[k] = TH[j]
            j = j + 1
```

Sorting Algorithms: Count Sort

Intuition:

"All these general-purpose algorithms are very nice. However, suppose all list-elements in L are positive integers and the maximum-value max in L is small? I could simply go through L once and count how many times elements of each value show up. All I have to do then is go through my count-list and print out each value the number of times it appears in L!"

Algorithm (Version #1):

```
Create array C of length max
Initialize all entries of C to zero
for each element L[i] in L do
   Increment C[L[i]] by one
for each element C[i] in C do
    if C[i] > 0 then
        Put C[i] copies of i in L
```

Sorting Algorithms: Count Sort

Algorithm (Version #2):

```
for i = 1 to max do
    C[i] = 0
for i = 1 to n do
    C[l[i]] = C[L[i]] + 1
pos = 1
for i = 1 to max do
    for j = 1 to C[i] do
        L[pos] = i
        pos = pos + 1
```

Sorting Algorithms Redux: Time Complexity

Merge Sort:

- Merge Sort is a **recursive algorithm**, *i.e.*, an algorithm that calls itself.
- The time complexities of recursive algorithms are described by special equations called **recurrences**. The recurrence for the time complexity of Merge Sort is as follows:

$$T(n) = \begin{cases} O(1) & n = 1 \\ 2T(n/2) + O(n) & n > 1 \end{cases}$$

  By math we will not go into here, this recurrence yields the time complexity $O(n \log_2 n)$.

Count Sort:

- Has loops that execute n and max times $\Rightarrow$ $O(n + max)$ time.
- Is only practical if max is small.

Sorting Algorithms Redux: Time Complexity (Cont'd)

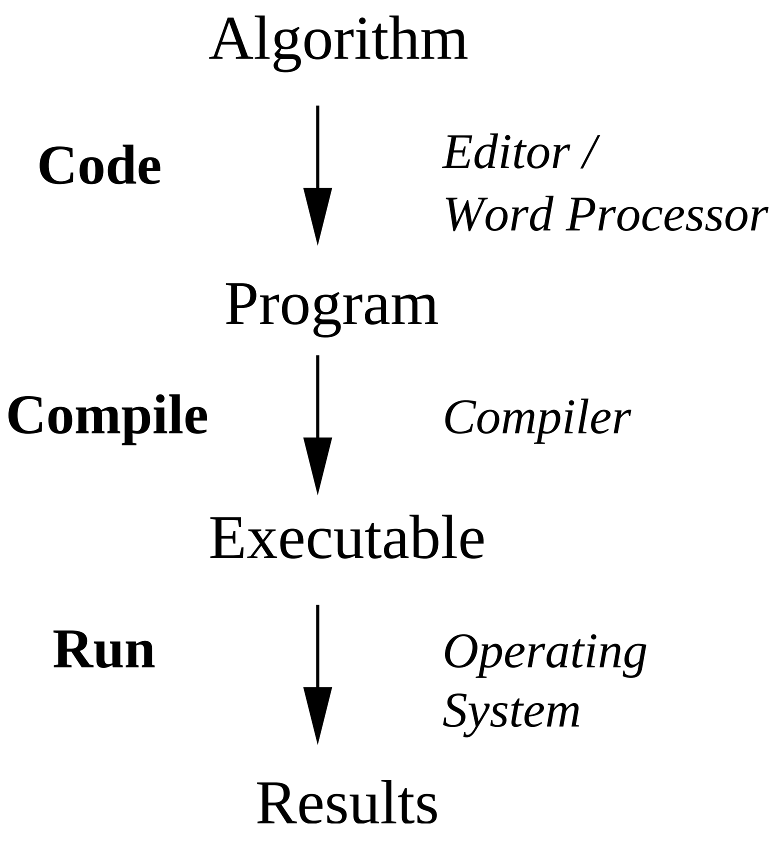
- Once again, on our megaflop computer . . .

	Time Complexity			
Input Size (n)	B-Search ($\log_2 n$)	L-Search, **C-Sort** (n)	**M-Sort** ($n \log_2 n$)	S/I/B-Sort (n^2)
10	.000003 second	.00001 second	.00003 second	.0001 second
20	.000004 second	.00002 second	.00008 second	.0004 second
30	.000005 second	.00003 second	.0001 second	.0009 second
50	.000006 second	.00005 second	.0003 second	.0025 second
100	.000007 second	.0001 second	.0007 second	.01 second
1000	.000010 second	.001 second	.01 second	1 second
10000	.000013 second	.01 second	.13 second	1.67 minutes
one million	.000020 second	1 second	20 seconds	11.57 days
300 million	.000028 second	5 minutes	2.34 hours	3 centuries
five billion	.000032 second	1.38 hours	1.86 days	7927 centuries

- Using the appropriate algorithm can make a *big* in how fast you can solve a problem!

Solving Problems with Programs: The Big Picture

- Four steps:
  1. Select / derive algorithm.
  2. Code algorithm as a program in some computer language.
  3. Compile program into an executable.
  4. Run executable on input data to create results.



Solving Problems with Programs: Count Sort in FORTRAN

```
      program csort

      integer  L(100), C(50)
      integer  i, j, n, max, pos;


*     Read in list L

      print *, "Number of elements?"
      read *, n

      print *, "Enter list:"
      do i = 1, n
          read *, L(i)
      end do

*     Find maximum value in list L

      max = 0
      do i = 1, n
          if (L(i).gt.max) then
              max = L(i)
          end if
      end do

*     Count-sort algorithm

      do i = 1, max
          C(i) = 0
      end do
      do i = 1, n
          C(L(i)) = C(L(i)) + 1
      end do
      pos = 1
      do i = 1, max
          do j = 1, C(i)
              L(pos) = i
              pos = pos + 1
          end do
      end do

*     Print out sorted version of list L

      print *, "Sorted list:"
      do i = 1, n
          print *, L(i)
      end do

      end
```

Solving Problems with Programs: Count Sort in FORTRAN (Cont'd)

```
Script started on Thu May  1 15:02:38 2003
complex% ls
csort.f  typescript
complex% f77 csort.f
complex% ls
a.out  csort.f  typescript
complex% a.out
 Number of elements?
5
 Enter list:
2
7
4
1
4
 Sorted list:
 1
 2
 4
 4
 7
complex% a.out
 Number of elements?
6
 Enter list:
1
3
2
10
9
4
 Sorted list:
 1
 2
 3
 4
 9
 10
complex% exit
Script done on Thu May  1 15:04:11 2003
```

Solving Problems with Programs: Count Sort in C

```
#include<stdio.h>


int main(int argc, char** argv){

    int  L[100], C[50];
    int  i, j, n, max, pos;
    char line[50];


    //  Read in list L

    printf("Number of elements?\n");
    gets(line);
    sscanf(line, "%d", &n);

    printf("Enter list:\n");
    for (i = 1; i <= n; i++) {
        gets(line);
        sscanf(line, "%d", &L[i]);
    }

    // Find maximum value in list L

    max = 0;
    for (i = 1; i <= n; i = i + 1) {
        if (L[i] > max) {
            max = L[i];
        }
    }

    // Count-sort algorithm

    for (i = 1; i <= max; i = i + 1) {
        C[i] = 0;
    }
    for (i = 1; i <= n; i = i + 1) {
        C[L[i]] = C[L[i]] + 1;
    }
    pos = 1;
    for (i = 1; i <= max; i = i + 1) {
        for (j = 1; j <= C[i]; j = j + 1) {
            L[pos] = i;
            pos = pos + 1;
        }
    }

    // Print out sorted version of list L

    printf("Sorted list:\n");
    for (i = 1; i <= n; i = i + 1) {
        printf("%d\n", L[i]);
    }

}  // End of main method
```

Solving Problems with Programs: Count Sort in C (Cont'd)

```
Script started on Thu May  1 15:09:32 2003
complex% ls
csort.c  typescript
complex% cc csort.c
/tmp/ccZu7fVL.o: In function `main':
/tmp/ccZu7fVL.o(.text+0x25): the `gets' function is dangerous and should not be used.
complex% ls
a.out  csort.c  typescript
complex% a.out
Number of elements?
5
Enter list:
2
7
4
1
4
Sorted list:
1
2
4
4
7
complex% a.out
Number of elements?
6
Enter list:
1
3
2
10
9
4
Sorted list:
1
2
3
4
9
10
complex% exit
Script done on Thu May  1 15:10:34 2003
```

Solving Problems with Programs: Count Sort in Java

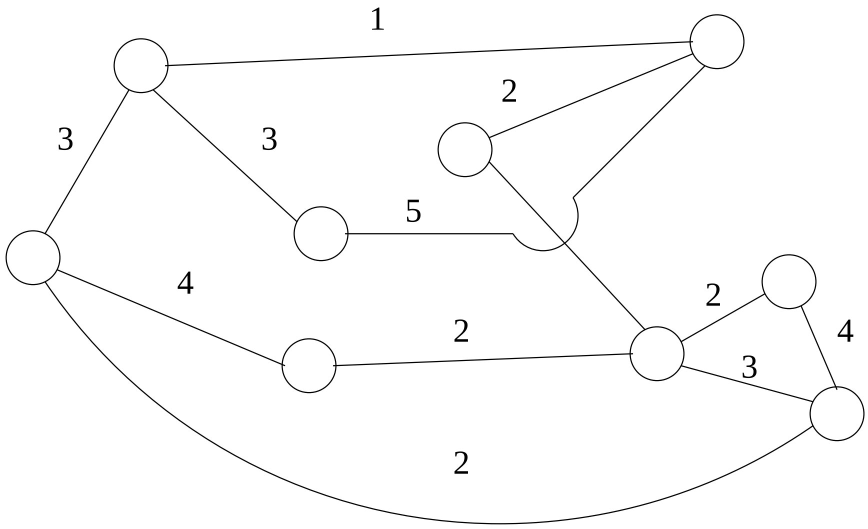
```
import java.io.*;


class csort {

    public static void main(String[] arg) throws IOException {

        BufferedReader stdin =
            new BufferedReader(new InputStreamReader(System.in));
        int[] L = new int[100], C = new int[50];
        int i, j, n, max, pos;


        //  Read in list L

        System.out.println("Number of elements?");
        n = Integer.parseInt(stdin.readLine());

        System.out.println("Enter list:");
        for (i = 1; i <= n; i++) {
            L[i] = Integer.parseInt(stdin.readLine());
        }

        // Find maximum value in list L

        max = 0;
        for (i = 1; i <= n; i = i + 1) {
            if (L[i] > max) {
                ‘max = L[i];
            }
        }

        // Count-sort algorithm

        for (i = 1; i <= max; i = i + 1) {
            C[i] = 0;
        }
        for (i = 1; i <= n; i = i + 1) {
            C[L[i]] = C[L[i]] + 1;
        }
        pos = 1;
        for (i = 1; i <= max; i = i + 1) {
            for (j = 1; j <= C[i]; j = j + 1) {
                L[pos] = i;
                pos = pos + 1;
            }
        }

        // Print out sorted version of list L

        System.out.println("Sorted list:");
        for (i = 1; i <= n; i = i + 1) {
            System.out.println(L[i]);
        }

    }  // End of main method

}  // End of class csort
```

Solving Problems with Programs: Count Sort in Java (Cont'd)

```
Script started on Thu May  1 16:25:36 2003
complex% ls
csort.java  typescript
complex% javac csort.java
complex% ls
csort.class  csort.java  typescript
complex% java csort
Number of elements?
5
Enter list:
2
7
4
1
4
Sorted list:
1
2
4
4
7
complex% java csort
Number of elements?
6
Enter list:
1
3
2
10
9
4
Sorted list:
1
2
3
4
9
10
complex% exit
Script done on Thu May  1 16:26:40 2003
```

Graph Problems: What are Graphs?

- A **graph** $G = (V, E)$ is a set of **vertices** V and a set of **edges** E that link pairs of vertices. Each edge may have an associated number (**weight**).



- Graphs are good at representing objects (vertices) and relationships between objects (edges), *e.g.*,
  - V = people, E = friendships between people
  - V = university courses, E = pairs of courses that have students in common
  - V = cities, E = roads between cities, edge weights = distances
  - V = houses, E = possible digital cable lines, edge weights = costs of cables

Graph Problems: Minimum Spanning Trees

- A **tree** is a graph without looping paths.

MINIMUM SPANNING TREE (MST) **Input**: An edge-weighted graph $G = (V, E)$. **Output**: A tree in G that connects all vertices in G and whose summed edge-weight is minimal.

- Has applications in designing communication networks.
- The simplest known algorithm sorts the edges by weight and adds edges starting with the lowest weight, leaving off those edges that form looping paths, until a tree is formed on all vertices in the graph – oddly enough, such a tree is guaranteed to have minimal summed edge-weight! This "greedy strategy" also works if you start with an arbitrary vertex and build the tree by always adding an edge with the smallest possible weight that connects a new vertex to the tree.
- Time complexity: $O(|V|^3)$

Graph Problems: Minimum Spanning Trees: Time Complexity

- Once again, on our megaflop computer ...

	Time Complexity				
Input Size (n)	B-Search ($\log_2 n$)	L-Search, C-Sort (n)	M-Sort ($n \log_2 n$)	S/I/B-Sort (n^2)	**MST** (n^3)
10	.000003 second	.00001 second	.00003 second	.0001 second	.001 second
20	.000004 second	.00002 second	.00008 second	.0004 second	.008 second
30	.000005 second	.00003 second	.0001 second	.0009 second	.027 second
50	.000006 second	.00005 second	.0003 second	.0025 second	.125 second
100	.000007 second	.0001 second	.0007 second	.01 second	1 second
1000	.000010 second	.001 second	.01 second	1 second	16.67 minutes
10000	.000013 second	.01 second	.13 second	1.67 minutes	11.57 days
one million	.000020 second	1 second	20 seconds	11.57 days	317 centuries
300 million	.000028 second	5 minutes	2.34 hours	3 centuries	9×10^9 centuries
five billion	.000032 second	1.38 hours	1.86 days	7927 centuries	4×10^{13} centuries

- This is starting to look bad; however, there are *much* harder problems out there ...

Graph Problems: Graph Coloring

GRAPH k-COLORING (k-COL)
Input: A graph $G = (V, E)$, a number $k \geq 1$.
Output: Is it possible to color the vertices of G with k colors such that no edge in G connects vertices with the same color?

- Has applications in scheduling groups of events (V = events, k = time slots).
- When $k = 2$, any vertex with one color requires that all vertices connected to it have the other color. This suggests a simple algorithm that fixes the color at an arbitrary vertex in G and spreads colors outwards in an alternating fashion to the rest of G; if this strategy ever forces an edge to connect two vertices of the same color, the graph requires more than two colors $\Rightarrow O(|E|) = O(|V|^2)$ time.
- When $k = 3$, there does not appear be an efficient algorithm for solving this problem. Indeed, the best known algorithms essentially look at all possible $3^{|V|}$ colorings of the graph to see if any one of them satisfies the problem conditions $\Rightarrow O(3^{|V|})$ time.

Graph Problems: Graph Coloring: Time Complexity

- Once again, on our megaflop computer . . .

Input	Time Complexity					
Input Size (n)	B-Search ($\log_2 n$)	L-Search, C-Sort (n)	M-Sort ($n \log_2 n$)	S/I/B-Sort, **2-Col** (n^2)	MST (n^3)	**3-Col** (3^n)
10	.000003 second	.00001 second	.00003 second	.0001 second	.001 second	.06 second
20	.000004 second	.00002 second	.00008 second	.0004 second	.008 second	58 minutes
30	.000005 second	.00003 second	.0001 second	.0009 second	.027 second	7 years
50	.000006 second	.00005 second	.0003 second	.0025 second	.125 second	2×10^8 centuries
100	.000007 second	.0001 second	.0007 second	.01 second	1 second	2×10^{32} centuries
1000	.000010 second	.001 second	.01 second	1 second	16.67 minutes	–
10000	.000013 second	.01 second	.13 second	1.67 minutes	11.57 days	–
one million	.000020 second	1 second	20 seconds	11.57 days	317 centuries	–
300 million	.000028 second	5 minutes	2.34 hours	3 centuries	9×10^9 centuries	–
five billion	.000032 second	1.38 hours	1.86 days	7927 centuries	4×10^{13} centuries	–

- Can we do better? Nobody knows. Perhaps all we need is a radically different kind of algorithm . . .

Conclusions

- Every algorithm is based on a simple intuition that is refined over several successively more specific versions into the finished algorithm.
- Ideally, you should verify your algorithm by tracing it on paper before you implement it as a program.
- Algorithms are rarely both simple and efficient. When selecting the best algorithm for an application, you need to figure out whether simplicity or efficiency is more important and then choose accordingly.
- There is still lots of work to be done ...

References

Computer Science 2710 (Problem Solving & Programming) Page:
`http://www.cs.mun.ca/~harold/Courses/List/CS2710.html`

Computer Science 3711 (Algorithms and Complexity) Page:
`http://www.cs.mun.ca/~harold/Courses/List/CS3711.html`

Computer Science 3718 (Programming in the Small) Page:
`http://www.cs.mun.ca/~harold/Courses/List/CS3718.html`

Computer Science Unplugged Home Page:
`http://unplugged.canterbury.ac.nz`