Algorithm Analysis
part 1

cse2011

Introduction

- What is an algorithm?
  - a clearly specified set of simple instructions to be followed to solve a problem
    - Takes a set of values, as input and produces a value, or set of values, as output
  - May be specified
    - In English
    - As a computer program
    - As a pseudo-code
- Data structures
  - Methods of organizing data
- Program = algorithms + data structures

Introduction

- Why we need algorithm analysis?
  - Writing a working program is not good enough.
  - The program may be inefficient!
  - If the program is run on a large data set, then the running time becomes an issue.

Example: Selection Problem

- Given a list of ***N*** numbers, determine the ***k***$^{\text{th}}$ largest, where $k \leq N$.
- Algorithm 1:

  (1) Read ***N*** numbers into an array

  (2) Sort the array in decreasing order by some simple algorithm

  (3) Return the element in position ***k***

Example: Selection Problem (2)

- Algorithm 2:
  (1) Read the first k elements into an array and sort them in decreasing order

  (2) Each remaining element is read one by one
    - If smaller than the k^{th} element, then it is ignored
    - Otherwise, it is placed in its correct spot in the array, bumping one element out of the array.

  (3) The element in the k^{th} position is returned as the answer.

Example: Selection Problem (3)

- Which algorithm is better when
  - **N** = 100 and **k** = 100?
  - **N** = 100 and **k** = 1?

- What happens when **N** = 1,000,000 and **k** = 500,000?

Algorithm Analysis

- We only analyze correct algorithms.
- An algorithm is correct
  - If, for **every input** instance, it halts (i.e., terminates) with the **correct output**.
- Incorrect algorithms
  - Might not halt at all on some input instances.
  - Might halt with other than the desired answer (i.e., the wrong answer).

Algorithm Analysis (2)

- Analyzing an algorithm
  - Predicting the resources that the algorithm requires.
  - Resources include
    - Memory (space)
    - Computational time (usually most important)
    - Communication bandwidth (in parallel and distributed computing)

Algorithm Analysis (3)

- Factors affecting the running time:
  - computer
  - compiler
  - algorithm used
  - input to the algorithm
    - The content of the input affects the running time
    - Typically, the input size (number of items in the input) is the main consideration.
      - sorting problem $\Rightarrow$ the number of items to be sorted
      - multiply two matrices together $\Rightarrow$ the total number of elements in the two matrices
    - And sometimes the input order as well (e.g., sorting algorithms).
- Machine model assumed
  - Instructions are executed one after another, with no concurrent operations $\Rightarrow$ not parallel computers

Analysis Model

- It takes exactly one time unit to do any calculation such as
  - + , -, * , /, %, &, |, &&, ||, etc.
  - comparison
  - assignment
- There is an infinite amount of memory.
- It does not consider the cost associated with page faulting or swapping.
- It does not include I/O costs (which is usually one or more orders of magnitude higher than computation costs).

An Example

```
int sum ( int n ) {
     int partialSum;
1    partialSum = 0;
2    for ( int i = 0; i <= n-1; i++ )
3         partialSum += i*i*i;
4    return partialSum;
}
```

- Lines **1** and **4**: one unit each
- Line **3**: 4N
- Line **2**: 1+(N+1)+N=2N+2
- Total: 6N+4 $\Rightarrow$ O(N)

Running Time Calculations

- Throw away leading constants.
- Throw away low-order terms.
- Compute a Big-Oh running time:
  – An upper bound for running time
  – Never underestimate the running time of a program
  – The program may end earlier, but never later (worst-case running time)

General Rules for Big-Oh: *for* loops

- *for* loops
  - at most the running time of the statements inside the *for* loop (including tests) times the number of iterations.
- Nested *for* loops

```
for (i = 0; i < n; i++ )
      for (j = 0; j < n; j++ )
            k ++;
```

  - the running time of the statement multiplied by the product of the sizes of all the ***for*** loops.
  - $O(N^2)$

Consecutive Statements

- Consecutive statements

```
for (i = 0; i < n; i++)
         a[i] = 0;
for (i = 0; i < n; i++)
         for (j = 0; j < n; j++)
                    a[i] += a[j] + i + j;
```

  - These just add.
  - **$O(N) + O(N^2) = O(N^2)$**

if – then – else

- ```
  if C then
     S1
  else
     S2
  ```
  - never more than the running time of the test plus the larger of the running times of S1 and S2.

```
if (n > 0)
   for ( int i = 0; i < n; i++ )
      sum += i;
else
   System.out.println( "Invalid input" );
```
```

Strategies

- Analyze from the inside out (loops).
- If there are method calls, analyze these first.
- Recursive methods (later):
  - Could be just a hidden “for” loop $\Rightarrow$ simple.
  - Solve a recurrence $\Rightarrow$ more complex.

Worst- / Average- / Best-Case

- Worst-case running time of an algorithm:
  - The **longest** running time for **any** input of size ***n***
  - An upper bound on the running time for any input

    $\Rightarrow$ guarantee that the algorithm will never take longer
  - Example: Sort a set of numbers in increasing order; and the input is in decreasing order
  - The worst case can occur fairly often
    - Example: searching a database for a particular piece of information
- Best-case running time:
  - sort a set of numbers in increasing order; and the input is already in increasing order
- Average-case running time:
  - May be difficult to define what “average” means

Example

- Given an array of integers, return true if the array contains number 100, and false otherwise.
  - Best case: ?
  - Worst case: ?
  - Average case: ?

Informal Introduction to O, Ω and Θ

- Given an unsorted array of integers, return true if a number k is in the array and false otherwise.

```
for( i = 0; i < N; i++ )
   if ( k == A[i] )
      return ( true );
return ( false );
```

- **Worst-case running time is O(N).**

  ⇒The alg. has O(N) running time.
- **Best-case running time is O(1).**

  ⇒ The alg. has Ω(1) running time.

- Given an unsorted array of integers, find and return the maximum value stored in the array.

```
max = A[0];
for( i = 1; i < N; i++ )
   if ( max < A[i] )
      max = A[i];
return( max );
```

- **Worst-case running time is O(N).**

  ⇒The alg. has O(N) running time.
- **Best-case running time is O(N).**

  ⇒The alg. has Ω(N) running time.

- **⇒The alg has Θ(N) running time.**

Running Time of Algorithms

- Bounds are for algorithms, rather than programs.
  - Programs are just implementations of an algorithm.
  - Almost always the details of the program do not affect the bounds.

- Bounds are for algorithms, rather than problems.
  - A problem can be solved with several algorithms, some are more efficient than others.

Example: Insertion Sort

1) Initially p = 1

2) Let the first p elements be sorted.

3) Insert the (p+1)th element properly in the list so that now p+1 elements are sorted.

4) Increment p and go to step (3)

Insertion Sort: Example

Original	34	8	64	51	32	21	Positions Moved
After $p = 1$	8	34	64	51	32	21	1
After $p = 2$	8	34	64	51	32	21	0
After $p = 3$	8	34	51	64	32	21	1
After $p = 4$	8	32	34	51	64	21	3
After $p = 5$	8	21	32	34	51	64	4

Insertion Sort: Algorithm

```
for(int p=1; p < a.size(); p++ ){
     int tmp=a[p];
     for(int j=p; j>0 && tmp<a[j-1]; j--)
          a[j] = a[j-1];
     a[j] = tmp;
}
```

- Consists of N - 1 passes
- For pass p = 1 through N - 1, ensures that the elements in positions 0 through p are in sorted order
  - elements in positions 0 through p - 1 are already sorted
  - move the element in position p left until its correct place is found among the first p + 1 elements

Example 2

To sort the following numbers in increasing order:

34   8   64   51   32   21

p = 1; tmp = 8;

34 > tmp, so second element a[1] is set to 34: {8, 34}…

We have reached the front of the list. Thus, 1st position a[0] = tmp=8

After 1st pass: 8   34   64   51 32   21

(first 2 elements are sorted)

P = 2; tmp = 64;

34 < 64, so stop at 3rd position and set 3rd position = 64

After 2nd pass: 8 34 64 51 32 21

(first 3 elements are sorted)

P = 3; tmp = 51;

51 < 64, so we have 8 34 64 64 32 21,

34 < 51, so stop at 2nd position, set 3rd position = tmp,

After 3rd pass: 8 34 51 64 32 21

(first 4 elements are sorted)

P = 4; tmp = 32,

32 < 64, so 8 34 51 64 64 21,

32 < 51, so 8 34 51 51 64 21,

next 32 < 34, so 8 34 34, 51 64 21,

next 32 > 8, so stop at 1st position and set 2nd position = 32,

After 4th pass: 8 32 34 51 64 21

P = 5; tmp = 21, . . .

After 5th pass: 8 21 32 34 51 64

Analysis: Worst-case Running Time

- What is the worst input?

- Consider a reversed sorted list as input.
- When a[p] is inserted into the sorted sub-array a[0...p-1], we need to compare a[p] with all elements in a[0...p-1] and move each element one position to the right

  $\Rightarrow$ i steps.

- Inner loop is executed p times, for each p = 1, 2, , ..., N-1

  $\Rightarrow$ Overall: $1 + 2 + 3 + \ldots + N\text{-}1 = \ldots = O(N^2)$

Analysis: Best-case Running Time

- The input is already sorted in the right order.
- When inserting a[p] into the sorted sub-array a[0...p-1], only need to compare a[p] with a[p-1] and there is no data movement

  $\Rightarrow$ O(1)

- For each iteration of the outer for-loop, the inner for-loop terminates after checking the loop condition once

  $\Rightarrow$ O(N) time

- If input is *nearly sorted*, insertion sort runs fast.

Insertion Sort: Summary

```
for(int p=1; p < a.size(); p++ ){
  int tmp=a[p];
     for(int j=p; j>0 && tmp<a[j-1]; j--)
          a[j] = a[j-1];
  a[j] = tmp;
}
```

- **$O(N^2)$**
- **$\Omega(N)$**
- Space requirement is O(?)

Next time …

- Growth rates
- O, Ω, Θ, o

- Reading for this lecture: chapter 4