CSc 225
Algorithms and Data Structures I
Case Studies

Jianping Pan

Fall 2007

Things we have so far

- Algorithm analysis
  - pseudo code
  - primitive operations
  - worst-case scenarios
- Asymptotic notations
  - Big-Oh

Find an Algorithm to solve Prefix Averages Problem

- Prefix Averages

Input: An n-element array X of numbers.
Output: An n-element array A of numbers such that $A[k]$ is the average of elements $X[0], \ldots, X[k]$

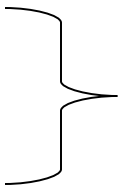
X	12	3	7	24	4	1	1
A	12	7.5	7.3	11.5	10	8.5	7.4

Algorithm PrefixAverages

Input: An n-element array X of numbers.

Output: An n-element array A of numbers such that $A[k]$ is the average of elements $X[0], \ldots, X[k]$

```
Let A be an array of n numbers.
for k←0 to n-1 do
   a←0
   for j←0 to k do
    a←a+X[j]
   end
   A[k]←a/(k+1)
end
return A
```



$k + 1$ times

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

Worst-Case Running Time of Algorithm PrefixAverages

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

$$\sum_{i=1}^{n} k=\frac{1}{2}n(n+1) \text{ is } O(n^2)$$

Algorithm PrefixAverages2

Input: An n-element array X of numbers.

Output: An n-element array A of numbers such that $A[k]$ is the average of elements $X[0], \ldots, X[k]$

```
Let A be an array of n numbers.
s←0
for k←0 to n−1 do
   s←s+X[k]
   A[k]← s/(k+1)
end
return A
```

n times

$O(n)$

PrefixAverages vs. PrefixAverages2

- PrefixAverages runs in *quadratic* time $O(n^2)$
- PrefixAverages2 runs in *linear* time $O(n)$

- Thus, PrefixAverages2 is more efficient!
- The analysis drove the design of PrefixAverages2 to a certain extent

Correctness of Algorithms

- One an algorithm has been properly specified, one can prove its *correctness*.
- That is the algorithm yields a required result for every legitimate input in a finite amount of time (i.e., it halts with the correct output).
  For some algorithms, a correctness proof is easy to obtain; for others it is very complex
- Mathematical induction is a common approach in proofing correctness
- Tracing the performance as in computing asymptotic complexity does not proof correctness

Optimality

- When is an algorithm optimal with respect to efficiency, complexity of implementation, maintainability, modifiability, extensibility, etc.?
- There are tradeoffs among all these criteria when implementing an algorithm
- Efficiency: What is the minimum amount of effort *any* algorithm will need to solve the given problem?
- For some problems we know the answer
  - Sorting $\Omega(n \log n)$
  - Searching $\Omega(1)$
  - Selection $\Omega(n)$

Implementing an Algorithm

- Most algorithms are eventually implemented as computer programs written in a programming language (e.g., coding in Java, C++, C#)
- In the transition from pseudo-code to code, correctness, efficiency and ease of understanding may be lost
- Formal verification is limited to small programs
- The validity of an implementation of an algorithm is established by testing
- Program testing can be used very effectively to show the presence of bugs but never to show their absence.

  E.W. Dijkstra, EWD 303

  http://www.cs.utexas.edu/~EWD/transcriptions/EWD03xx/EWD303.html

How can we prove correctness of an algorithm? (Read Section 1.3.3)

- Using standard proof techniques like
  - Induction
  - Contra attack
  - Counterexample
  - Loop invariants

Running time of recursiveMax

```
Algorithm recursiveMax(A,n):
  Input: An array A storing n≥1 integers.
  Output: The maximum element in A.

  if n = 1 then
   return A[0]
  return max{recursiveMax(A,n−1),
  A[n−1]}
```

if $n = 1$

and running time of recursiveMax(A, $n-1$) if $n > 1$

Running time of recursiveMax

$$T(n) = \begin{cases} 3 & \text{if } n = 1 \\ T(n-1) + 9 & \text{otherwise} \end{cases}$$

Running time of recursiveMax for large n

$$T(n) = T(n-1) + 9$$
$$= T(n-2) + 9 + 9 = T(n-2) + 2 \cdot 9$$
$$= T(n-3) + 9 + 2 \cdot 9 = T(n-3) + 3 \cdot 9$$
$$\vdots$$
$$= T(n-i) + 9 + (i-1) \cdot 9 = T(n-i) + i \cdot 9$$

We know: $T(n - i) = 3$ iff $n - i = 1$ iff $n - 1 = i$.

Thus $T(n) = T(n - i) + i \cdot 9 = T(n - (n - 1)) + (n - 1)9$
$= T(1) + 9n - 9 = 3 + 9n - 9 = 9n - 6$

Correctness of recursiveMax

Proof by induction on the number of elements in the array.

```
Algorithm recursiveMax(A,n):
   Input: An array A storing n≥1
    integers.
   Output: The maximum element in A.

   if n = 1 then
    return A[0]
   return max{recursiveMax(A,n−1),
   A[n−1]}
```

Base step: $n = 1$
$A[0]$ is the only element and therefore the maximum.

Hypothesis: recursiveMax is correct for n (i.e., recursiveMax(A, $n-1$) returns the maximum element of the first n elements in A.)

Correctness of recursiveMax

Proof by induction on the number of elements in the array.

```
Algorithm recursiveMax(A,n):
   Input: An array A storing n≥1
    integers.
   Output: The maximum element in A.

   if n = 1 then
    return A[0] end
   return max{recursiveMax(A,n–1),
   A[n–1]}
```

Base step: $n = 1$
$A[0]$ is the only element and therefore the maximum.

Induction step: Show recursiveMax is correct for $n + 1$.
max{recursiveMax(A,n), $A[n]$}

This lecture

- Algorithm analysis case studies
  - correctiveness
  - performance
    - asymptotic analysis
- Explore further

Next lecture

- Data structures
  - read AD Chapter 2