Analysis of Algorithms

T. M. Murali

Mar 24, 2004
Review

- Inheritance can provide shared implementation.
- Abstract classes.
  - Abstract methods allow static type checking without requiring implementation.
  - Abstract classes function as incomplete superclasses.
  - You cannot create an instance of an abstract class.
  - Abstract classes support polymorphism.
- Interfaces
  - Provide specification without implementation.
  - Interfaces are fully abstract.
  - Interfaces support polymorphism.
  - Java interfaces support multiple inheritance.
Analysis of Algorithms

- An algorithm is a step-by-step procedure for solving a problem in a finite amount of time.
- Most algorithms transform input objects into output objects.
- The running time of an algorithm typically grows with the input size.
- Average case time is often difficult to determine.
- We focus on the worst case running time.
Experimental Analysis

- Write a program implementing the algorithm.
- Run the program with inputs of varying size and difficulty.
- Measure the actual running time.
- Plot the results.
Limitations of Experiments

- It is necessary to implement the algorithm, which may be difficult.
- Results may not be indicative of the running time on inputs not included in the experiment.
- In order to compare two algorithms, the same hardware and software environments must be used.
Theoretical Analysis

- Uses a high-level description of the algorithm instead of an implementation.
- Characterizes running time as a function of the input size, $n$.
- Takes into account all possible inputs.
- Allows us to evaluate the speed of an algorithm independent of the hardware/software environment.
Pseudocode

- High-level description of an algorithm.
- More structured than a textual description but less detailed than an implementation.
- Commonly-used notation for describing algorithms.
- Hides program design issues.

Algorithm arrayMax(A, n):

**Input:** An array A storing \( n \geq 1 \) numbers.

**Output:** The maximum element in A.

\[
\text{currentMax} \leftarrow A[0];
\]

\[
\text{for } i \leftarrow 1 \text{ to } n - 1 \text{ do}
\]

\[
\text{if } \text{currentMax} \leq A[i] \text{ then}
\]

\[
\text{currentMax} = A[i]
\]

\[
\text{return } \text{currentMax}
\]
Random Access Memory Model of Computation

- A computer consists of a CPU that can perform certain simple operations (e.g., addition, control, assignment) in unit time.
- A memory unit. Accessing any single element of memory (e.g., a variable or an array index) takes constant time.
Primitive Operations

- Set of basic computations that an algorithm performs.
- Almost all programming languages have these operations.
- Take a constant amount of time to execute in the RAM model.
- Examples:
  - Evaluating an expression
  - Assigning a value to a variable
  - Indexing into an array
  - Calling a method
  - Returning from a method
Counting Primitive Operations

Inspect the pseudocode to count maximum number of primitive operations executed by an algorithm, as a function of the input size.

Algorithm arrayMax(A, n):

Input: An array A storing $n \geq 1$ numbers.
Output: The maximum element in A.

1. $currentMax \leftarrow A[0]$
2. for $i \leftarrow 1$ to $n - 1$ do
   1. if $currentMax < A[i]$ then
      1. $currentMax = A[i]$
3. return $currentMax$
Estimating Running Time

- Algorithm arrayMax executes $8n - 2$ primitive operations in the worst case. Suppose
  - $a =$ time taken by the fastest primitive operation
  - $b =$ time taken by the slowest primitive operation

- If $T(n)$ is worst-case running time of arrayMax on an array with $n$ indices, then
  \[
  a(8n - 2) \leq T(n) \leq b(8n - 2)
  \]

- The running time $T(n)$ is bounded by two linear functions.
- Changing software or hardware only change $a$ and $b$.
- Linear growth rate of $T(n)$ is an intrinsic property of arrayMax.
Affect of Constant Factors

- The growth rate is not affected by
  - constant factors or
  - lower-order terms

- Example:
  - $158n + 10^5$ is a linear function
  - $44.3n^2 + 10^7 n$ is a quadratic function.
Big-Oh Notation

- Given functions $f(n)$ and $g(n)$, we say that $f(n)$ is $O(g(n))$ (pronounced “order of” or “Big-oh”) if there are positive constants $c$ and $n_0$ such that

$$f(n) \leq cg(n), \text{ for all } n \geq n_0$$

- Examples:
  - $158n + 10^5$ is $O(n)$.
  - $44n^2 + 10^7 n$ is $O(n^2)$
  - BUT $n^2$ is not $O(n)$
  - What about $5 \log n$? Is it $O(n^2)$, $O(n)$, or $O(\log n)$?
Big-Oh and Growth Rate

- The big-Oh notation gives an (asymptotic) upper bound on the growth rate of a function.
- The statement $f(n)$ is $O(g(n))$ says that the growth rate of $f(n)$ is no more than the growth rate of $g(n)$.
- We can use the big-Oh notation to rank functions according to their growth rate.
- We can use the big-Oh notation to rank algorithms according to their running time.
  - An algorithm that runs in $O(n)$ time is better than an algorithm that runs in $O(n^2)$ time.
Useful Properties of Big-Oh

- If \( f(n) \) is a polynomial of degree \( d \), then \( f(n) = O(n^d) \).
- Use the smallest possible function inside \( O() \).
  - We say \( 2n \) is \( O(n) \) rather than \( O(n^2) \).
- Use the simplest expression in a class (\( O() \) eats up all constant factors).
  - We say that \( 2n + 3 \) is \( O(n) \) rather than \( O(2n) \).
Asymptotic Analysis of Algorithms

- The asymptotic analysis of an algorithm measures the running time as the size of the input becomes very large.
- Allows us to express running time using big-Oh notation.
- Allows us to ignore constant factors and lower order terms.
  - We can disregard these terms when counting primitive operations.
- Example:
  - arrayMax executes at most $8n - 2$ primitive operations.
  - Therefore, arrayMax runs in $O(n)$ time.
Another Example: arrayMinMax

Algorithm arrayMinMax($A$, $n$):

**Input:** An array $A$ storing $n \geq 1$ numbers.

**Output:** The minimum and maximum elements in $A$.

```
currentMin ← $A[0]$

currentMax ← $A[0]$

for $i ← 1$ to $n − 1$ do
    if $currentMax < A[i]$ then
        $currentMax = A[i]$
    else if $currentMin > A[i]$ then
        $currentMin = A[i]$

return $currentMin$, $currentMax$
```

What is the running time of arrayMinMax?

The running time is $O(n)$. 

Another Example: arrayMinMax

Algorithm arrayMinMax(A, n):

Input: An array A storing $n \geq 1$ numbers.
Output: The minimum and maximum elements in A.

$\text{currentMin} \leftarrow A[0]$;
$\text{currentMax} \leftarrow A[0]$;
for $i \leftarrow 1$ to $n - 1$ do
  if $\text{currentMax} < A[i]$ then
    $\text{currentMax} = A[i]$
  else if $\text{currentMin} > A[i]$ then
    $\text{currentMin} = A[i]$
return $\text{currentMin}, \text{currentMax}$

▶ What is the running time of arrayMinMax?

▶ The running time is $O(n)$. 
Computing Prefix Averages

Another example of asymptotic analysis.

The $i$th prefix average of an array $X$ is the average of the first $i + 1$ elements of $X$:

$$A[i] = \frac{X[0] + X[1] + \ldots + X[i]}{i + 1}$$
A PrefixAverages Algorithm

Algorithm prefixAverages1(X):

Input: An array X storing \( n \geq 1 \) numbers.

Output: An array A with \( n \) numbers such that \( A[i] \) is the \( i \)th prefix average of \( X \).

Let \( A \) be an array of \( n \) numbers.

for \( i \leftarrow 0 \) to \( n - 1 \) do

\[
\text{sum} \leftarrow 0
\]

for \( j \leftarrow 0 \) to \( i \) do

\[
\text{sum} = \text{sum} + X[j]
\]

\[
A[i] = \text{sum}/(i + 1).
\]

return array \( A \).

What is the running time of prefixAverages1?

The running time is \( O(n^2) \).
A PrefixAverages Algorithm

Algorithm prefixAverages1(X):

Input: An array X storing \( n \geq 1 \) numbers.
Output: An array A with n numbers such that \( A[i] \) is the ith prefix average of X.

Let A be an array of n numbers.

\[ \text{for } i \leftarrow 0 \text{ to } n - 1 \text{ do} \]
\[ \quad \text{sum} \leftarrow 0 \]
\[ \quad \text{for } j \leftarrow 0 \text{ to } i \text{ do} \]
\[ \quad \quad \text{sum} = \text{sum} + X[j] \]
\[ \quad A[i] = \text{sum} / (i + 1). \]

\text{return array A.}

What is the running time of prefixAverages1?

The running time is \( O(n^2) \).
Faster Algorithm

Algorithm prefixAverages2(X):

Input: An array X storing $n \geq 1$ numbers.

Output: An array A with n numbers such that $A[i]$ is the $i$th prefix average of $X$.

Let $A$ be an array of $n$ numbers.

$sum = 0$

for $i \leftarrow 0$ to $n - 1$ do

$sum = sum + X[i]$

$A[i] = sum/(i + 1)$.

return array A.

What is the running time of prefixAverages2?

The running time is $O(n)$. 
Faster Algorithm

Algorithm prefixAverages2($X$):

**Input:** An array $X$ storing $n \geq 1$ numbers.

**Output:** An array $A$ with $n$ numbers such that $A[i]$ is the $i$th prefix average of $X$.

Let $A$ be an array of $n$ numbers.

$sum = 0$

for $i \leftarrow 0$ to $n - 1$ do

$sum = sum + X[i]$

$A[i] = sum/(i + 1)$.

return array $A$.

What is the running time of prefixAverages2?

The running time is $O(n)$. 

▶
Friends of Big-Oh

- **Big-Omega (lower-bound)**
  - \( f(n) \) is \( \Omega(g(n)) \) if there are positive constants \( c \) and \( n_0 \) such that
    \[
    f(n) \geq cg(n), \text{ for } n \geq n_0
    \]

- **Big-Theta**
  - \( f(n) \) is \( \Theta(g(n)) \) if \( f(n) \) is \( O(g(n)) \) and \( f(n) \) is \( \Omega(g(n)) \).

- **Examples**
  - \( 10n^2 \) is \( \Omega(n^2) \) (is also \( \Omega(n) \), but \( \Omega(n^2) \) is a “tighter” lower bound.
  - \( 10n^2 \) is \( O(n^2) \).
  - \( 10n^2 \) is \( \Theta(n^2) \).