Lecture 3: Algorithm Analysis

Bohyung Han
CSE, POSTECH
bhhan@postech.ac.kr

Algorithm

• A step-by-step procedure to solve a problem
  ▪ Start from an initial state and input
  ▪ Proceed through a finite number of successive states
  ▪ Stop when reaching a final state and producing output

![Algorithm Diagram]

Algorithm Performance

• Algorithm performance is measured by the amount of computer memory and running time required to run an algorithm.

• Performance measurement
  ▪ Empirical analysis
    • Compute memory space in use and running time
    • Results are not general—valid only for tested inputs.
    • Same machine environment must be used to compare two algorithms.
  ▪ Theoretical analysis
    • Compute asymptotic bound of space and running time
    • Sometimes, it is difficult to measure average cases; worst case analysis are often used.
    • Machine independent

Empirical Analysis

• Measurement criteria
  ▪ Actual memory space in use
  ▪ Running time of algorithms

• Example

![Empirical Analysis Diagram]

Algorithm1 and 2 took 0.3 and 0.5 milliseconds for Input1, respectively.
Algorithm1 may be faster than Algorithm2.
Algorithm1 and 2 took 1.0 and 0.8 milliseconds for Input2, respectively.
How about now?
Theoretical Analysis

- Measurement criteria
  - Space: amount of memory in bytes that algorithm occupies
  - Time
    - Running time
    - Typically measured by the number of primitive operations

- Algorithm analysis
  - Use a pseudocode of the algorithm
  - Need to take all possible inputs into account
  - Evaluate the algorithm speed independent of the machine environment

Pseudocode

- High-level description of an algorithm
  - Independent of any programming language
  - More structured than English proses
  - Less detailed than program codes
  - Hiding program design issues
  - Easy to understand

Algorithm arrayMax(A, n)

Input array A of n integers
Output maximum element of A

currentMax \leftarrow A[0]
for \ i \ from 1 \ to \ n - 1 \ do
  if A[i] = currentMax then
    currentMax \leftarrow A[i]
return currentMax

Space Complexity

- Amount of necessary memory for an algorithm
  - The space complexity may define an upper bound on the data that the algorithm uses.

- Why do we care about space complexity?
  - We may not have sufficient memory space in our computer.
  - When we solve a large-scale problem, memory space is often a critical bottleneck.

Space Complexity

- Data space depends on
  - Computer architecture and compiler
  - Implementation

```c
double array1[100];
int array2[100];

// The size of array1 is twice bigger than that of array2.
```

- Algorithm: e.g., dense matrix vs. sparse matrix

<table>
<thead>
<tr>
<th>Dense matrix:</th>
<th>4</th>
<th>8</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Sparse matrix: (index, value)</td>
<td>(2,4), (5,8), (9,1)</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>6 integers</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
**Time Complexity**

- Amount of time required to run an algorithm

- Why do we care about time complexity?
  - Some applications require real-time responses.
  - If there are many solutions for a problem, we typically prefer the quickest one.

- How do we measure?
  - Count a particular operation (operation counts) e.g., comparisons in sorting
  - Count the number of steps (step counts)
  - Asymptotic complexity

**Example: Insertion Sort**

- How many comparisons are made?

```plaintext
Algorithm insertionSort(A, n)
Input array A of n integers
Output the sorted array A
for i <- 1 to n - 1 do
  t <- A[i]
  for j <- i-1 to 0 do
    else break
  A[j+1] <- t
return A
```

**Worst Case Analysis**

- Why using the worst case?
  - Average case is sometimes difficult to analyze.
  - The time complexity of the worst case is often very important.

- An example: insertion sort.
  - All elements are reversely sorted.
  - Total number of comparisons:
    
    \[ 1 + 2 + \cdots + n = \frac{n(n + 1)}{2} \]

**Primitive Operations**

- Basic computations performed by an algorithm
  - Constant time assumed for execution
  - Identifiable in pseudocode

- Largely independent from the programming language

- Examples
  - Assignments: a <- b
  - Comparisons: a < b
  - Arithmetic operations: a+b
  - Dereference: A[i]
  - Returning from a method
Primitive Operation Counts

- By inspecting the pseudocode, we can determine the maximum number of primitive operations executed by an algorithm, as a function of the input size.

<table>
<thead>
<tr>
<th>Algorithm insertionSort(A, n)</th>
<th># operations</th>
<th>total # operations</th>
</tr>
</thead>
<tbody>
<tr>
<td>Input array A of n integers</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Output the sorted array A</td>
<td></td>
<td></td>
</tr>
<tr>
<td>for i &lt;- 1 to n - 1 do</td>
<td>1</td>
<td>(n-1)</td>
</tr>
<tr>
<td>t &lt;- A[i]</td>
<td>2</td>
<td>2(n-1)</td>
</tr>
<tr>
<td>for j &lt;- i-1 to 0 do</td>
<td>1</td>
<td>0.5(n-1)n</td>
</tr>
<tr>
<td>else break</td>
<td></td>
<td></td>
</tr>
<tr>
<td>A[j+1] &lt;- t</td>
<td>2</td>
<td>2(n-1)</td>
</tr>
<tr>
<td>return A</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Total</td>
<td></td>
<td>4.5n^2 + 0.5n - 4</td>
</tr>
</tbody>
</table>

Running Time Analysis

- Running time, $T(n)$
  - Computational complexity of an algorithm in a theoretical model
  - Example
    - The total number of operations of insertion sort is $4.5n^2 + 0.5n - 4$.
    - The theoretical running time may be bounded by other functions:
      $$a(4.5n^2 + 0.5n - 4) \leq T(n) \leq b(4.5n^2 + 0.5n - 4),$$
      where $a$ and $b$ are times taken by the fastest and slowest primitive operations.

- Running time analysis
  - Growth function characterizes running time with respect to number of inputs.
  - We are interested in the growth rate of the running time.

Growth Functions

- Constant
  - Growth is independent of the input size, $n$
  - $T(n) = c$
  - e.g., accessing array element at known location, assignment

- Linear
  - Growth is directly proportional to $n$
  - $T(n) = cn$
  - e.g., finding a particular array element (linear search)

- Logarithmic
  - Growth increases slowly compared to $n$
  - $T(n) = \log(n)$
  - e.g., finding particular array element (binary search)

- Quadratic
  - $T(n) = n^2$
  - e.g., typical in nested loops (bubble sort)

- Cubic
  - $T(n) = n^3$
  - e.g., matrix multiplication

- Exponential
  - Growth is extremely rapid and possibly impractical
  - $T(n) = c^n$
  - e.g., Towers of Hanoi ($2^n$), Fibonacci number ($2^n$)
Growth Functions

- Other polynomial
  - \( T(n) = n^k \)
  - e.g., typical in more nested loops

- Others
  - \( T(n) = n \log(n) \)
  - e.g., sorting using divide and conquer approach

A Table of Growth Functions

<table>
<thead>
<tr>
<th>( \log(n) )</th>
<th>( n )</th>
<th>( n \log(n) )</th>
<th>( n^2 )</th>
<th>( n^3 )</th>
<th>( 2^n )</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td>2</td>
<td>4</td>
<td>8</td>
<td>4</td>
</tr>
<tr>
<td>2</td>
<td>4</td>
<td>8</td>
<td>16</td>
<td>64</td>
<td>16</td>
</tr>
<tr>
<td>3</td>
<td>8</td>
<td>24</td>
<td>64</td>
<td>512</td>
<td>256</td>
</tr>
<tr>
<td>4</td>
<td>16</td>
<td>64</td>
<td>256</td>
<td>4096</td>
<td>65536</td>
</tr>
<tr>
<td>5</td>
<td>32</td>
<td>160</td>
<td>1024</td>
<td>32768</td>
<td>4294967296</td>
</tr>
</tbody>
</table>

- Overly complex algorithm may be impractical in some system

Asymptotic Complexity

- Asymptotic
  - Definition: approaching a given value as an expression containing a variable tends to infinity

- Asymptotic complexity
  - Describe the growth rate of the time or space complexity with respect to input size
  - Is not directly related to the exact running time

- Three notations
  - Big \( O(\) notation provides an upper bound for the growth rate of the given function.
  - Big Omega \( (\Omega) \) notation provides a lower bound.
  - Big Theta \( (\Theta) \) notation is used when an algorithm is bounded both above and below by the same kind of function.
Big O

- \( f(n) \in O(T(n)) \)
  - \( \exists c, \exists n_0, \forall n > n_0 \)
  - \( f(n) \leq c \cdot T(n) \)
  - Note: \( c \) and \( n_0 \) may not be unique.

- Rule of thumb
  - Ignore less dominating term.
  - Ignore constant

- Example
  - \( (3n^2 - 5) \in O(n^2) \).
  - \( 3n^2 - 5 \leq 4n^2 \quad c = 4 \) and \( n_0 = 3 \)

More Big O Examples

- \( 7n - 2 \in O(n) \)
  - \( 7n - 2 \leq 7n \) for \( n \geq 1 \) (\( c = 7, n_0 = 1 \))

- \( 3n^3 + 20n^2 + 5 \in O(n^3) \)
  - \( 3n^3 + 20n^2 + 5 \leq 4n^3 \) for \( n \geq 21 \) (\( c = 4, n_0 = 21 \))

- \( 3 \log(n) + 5 \in O(\log(n)) \)
  - \( 3 \log(n) + 5 \leq 8 \log(n) \) for \( n \geq 2 \) (\( c = 8, n_0 = 2 \))

Big Omega

- \( f(n) \in \Omega(T(n)) \)
  - \( \exists c, \exists n_0, \forall n > n_0 \)
  - \( c \cdot T(n) \leq f(n) \)
  - Note: \( c \) and \( n_0 \) may not be unique.

- Example
  - \( (0.5n^2 - 1) \in \Omega(n^2) \)
  - \( 0.5n^2 - 1 \geq 0.4n^2 \)
  - \( c = 0.4 \) and \( n_0 = 0 \)

Big Theta

- \( f(n) \in \Theta(T(n)) \)
  - \( \exists c_1, \exists c_2, \exists n_0, \forall n > n_0 \)
  - \( c_1 \cdot T(n) \leq f(n) \leq c_2 \cdot T(n) \) for \( n \geq n_0 \)
  - Note: \( c_1, c_2 \) and \( n_0 \) may not be unique.

- The total number of primitive operations of an algorithm is \( n^2 + n - 1 \).
  - The upper bound is \( O(n^2) \)
  - The lower bound is \( \Omega(n^2) \)
  - Therefore, \( \Theta(n^2) \)
Caveat

- The relative behavior of two functions is compared only \textit{asymptotically}, for large \( n \).
- The asymptotic complexity may make no sense for small \( n \).

- Example: two algorithms with the same complexity, \( O(n) \)
  - Algorithm A executes 3 primitive operations in a loop.
  - Algorithm B executes 10 primitive operations in a loop.
  - Then the algorithm A is faster than the algorithm B although their complexities are same.

- Example: two algorithms with \( O(n) \) and \( O(n^2) \)
  - With a small number of inputs, algorithm with quadratic complexity may be faster than algorithm with linear complexity.
  - Think about the definition of Big O notation.

Example

```c
int Example1(int[][] a) {
    int sum = 0;
    for (int i=0; i<a.length; i++)
        for (int j=0; j<i; j++)
            sum = sum + a[i][j];
    return sum;
}
```

Number of operations

\[
0 + 1 + \cdots + (n - 1) = \frac{n(n - 1)}{2}
\]

\[\Rightarrow O(n^2)\]

\( n \): number of rows

Example

```c
int Example2(int[] a) {
    int sum = 0;
    for (int i=a.length; i>0; i/=2)
        sum = sum + a[i];
    return sum;
}
```

Number of operations

\[
\log_2 n + \log_2 n + \cdots + \log_2 n
\]

\[\Rightarrow O(n \log n)\]

\( n \): number of elements

Example

```c
int Example3(int[] a) {
    int sum = 0;
    for (int i=0; i<a.length; i++)
        for (int j=a.length; j>0; j/=2)
            sum = sum + a[i] + a[j];
    return sum;
}
```

Number of operations

\[
\log n + \log n + \cdots + \log n
\]

\[\Rightarrow O(n \log n)\]

\( n \): number of elements
Example

```java
int Example4(int[][] a)
{
    int sum = 0;
    for (int i=0; i<a.length; i++)
    {
        for (int j=0; j<100; j++)
        {
            sum = sum + a[i][j];
        }
    }
    return sum;
}
```

Number of operations:

$$100 + 100 \cdots + 100 = 100n$$

$$\Rightarrow O(n)$$

Example

```java
int Example5(int n)
{
    int sum1 = 0;
    int sum2 = 0;
    for (int i=0; i<n; i++)
    {
        sum1 += i;
        for (int j=10*n; j<100*n; j++)
        {
            sum2 += sum1;
        }
    }
    return sum;
}
```

Number of operations:

$$90n + 90n \cdots + 90n = 90n^2$$

$$\Rightarrow O(n^2)$$

Example

```java
int[] [] multiplySquareMatrices(int[][] a, int[][] b)
{
    int[][] c = new int[a.length][a.length];
    int n = a.length;
    for (int i=0; i<n; i++)
    {
        for (int j=0; j<n; j++)
        {
            for (int k=0; k<n; k++)
            {
                c[i][j] = c[i][j] + a[i][k] * b[k][j];
            }
        }
    }
    return c;
}
```

Example

```java
Dense matrix A:  
(4, 8, 1)

Sparse matrix A:  
(index, value)  
(2,4), (5,8), (9,1)
```

Trade-off

- Space complexity and time complexity may not be independent.
  - There is a trade-off between the two complexities.
  - Algorithm A may be faster than Algorithm B, but consume memory space more than Algorithm B.

<table>
<thead>
<tr>
<th></th>
<th>Dense matrix A</th>
<th>Sparse matrix A</th>
</tr>
</thead>
<tbody>
<tr>
<td>Space complexity</td>
<td>4*10 bytes</td>
<td>4*6 bytes</td>
</tr>
<tr>
<td>Time complexity of A[i]</td>
<td>O(1)</td>
<td>O(n)</td>
</tr>
</tbody>
</table>
Summary

- Complexity analysis mostly involves examining loops.
- Attempt to characterize growth of functions.
  - This will be important when we examine searching and sorting
- Big O notation helps us simplify complexity definitions – worst case
  - It ignores constant and lower order terms.
  - Big Omega (Ω) examines lower bounds.
  - Big Theta (Θ) examines specific case.
- Computer scientists often use proofs to defend complexity claims.