Fundamentals of Analysis of Algorithm Efficiency

Analysis of Algorithms

• Issues:
  • Correctness
  • Time efficiency
  • Space efficiency
  • Optimality

• Approaches:
  • Theoretical analysis
  • Empirical analysis
Theoretical Analysis of Time Efficiency

Time efficiency is analyzed by determining the number of repetitions of the basic operation as a function of input size.

- **Basic operation**: the operation that contributes most towards the running time of the algorithm.

$$T(n) \approx c_{op} C(n)$$

### Input Size and Basic Operation Examples

<table>
<thead>
<tr>
<th>Problem</th>
<th>Input size measure</th>
<th>Basic operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Search for key in list of $n$ items</td>
<td>Number of items in list, $n$</td>
<td>Key comparison</td>
</tr>
<tr>
<td>Multiply two matrices of floating point numbers</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Compute $a^n$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Checking primality of a given integer $n$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Graph problem</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Empirical Analysis of Time Efficiency

- Select a specific (typical) sample of inputs
- Use physical unit of time (e.g., milliseconds)
  OR
  Count actual number of basic operations
- Analyze the empirical data

Best-Case, Average-Case, Worst-Case

For some algorithms efficiency depends on form of input:

- Worst case: $C_{\text{worst}}(n)$ – maximum over inputs of size $n$
- Best case: $C_{\text{best}}(n)$ – minimum over inputs of size $n$
- Average case: $C_{\text{avg}}(n)$ – “average” over inputs of size $n$
  - Number of times the basic operation will be executed on typical input
  - NOT the average of worst and best case
  - Expected number of basic operations considered as a random variable under some assumption about the probability distribution of all possible inputs
Example: Sequential Search

- **Problem:** Given a list of \( n \) elements and a search key \( K \), find an element equal to \( K \), if any.
- **Algorithm:** Scan the list and compare its successive elements with \( K \) until either a matching element is found (successful search) of the list is exhausted (unsuccessful search)

```
ALGORITHM       SequentialSearch(A[0..n − 1], K)
//Searches for a given value in a given array by sequential search
//Input: An array A[0..n − 1] and a search key K
//Output: The index of the first element of A that matches K
// or −1 if there are no matching elements
i ← 0
while i < n and A[i] ≠ K do
    i ← i + 1
if i < n return i
else return −1
```

Example: Sequential Search

- **Worst case**
  - What is the worst case / what are the worst cases?
  - #basic operations, i.e., comparisons = ?

- **Best case**
  - What is the best case / what are the best cases?
  - #basic operations = ?

- **Average case**
  - How many cases are there?
  - Assume each case is equally likely to occur, #basic operations = ?
Types of Formulas for Basic Operation Count

- **Exact formula**
  
  e.g., \( C(n) = \frac{n(n-1)}{2} \)

- **Formula indicating order of growth with specific multiplicative constant**
  
  e.g., \( C(n) \approx 0.5 \, n^2 \)

- **Formula indicating order of growth with unknown multiplicative constant**
  
  e.g., \( C(n) \approx c \, n^2 \)

Order of Growth

- **Most important:** Order of growth within a constant multiple as \( n \to \infty \)

- **Example:**
  
  - How much faster will algorithm run on computer that is twice as fast?
  
  - How much longer does it take to solve problem of double input size?

- **See table 2.1**
### Table 2.1

<table>
<thead>
<tr>
<th>(n)</th>
<th>(\log_2 n)</th>
<th>(n)</th>
<th>(n \log_2 n)</th>
<th>(n^2)</th>
<th>(n^3)</th>
<th>(2^n)</th>
<th>(n!)</th>
</tr>
</thead>
<tbody>
<tr>
<td>(10)</td>
<td>3.3</td>
<td>(10^1)</td>
<td>3.3 (\cdot 10^1)</td>
<td>(10^2)</td>
<td>(10^3)</td>
<td>(10^3)</td>
<td>(3.6 \cdot 10^6)</td>
</tr>
<tr>
<td>(10^2)</td>
<td>6.0</td>
<td>(10^2)</td>
<td>0.6 (\cdot 10^2)</td>
<td>(10^4)</td>
<td>(10^6)</td>
<td>(1.3 \cdot 10^5)</td>
<td>(9.3 \cdot 10^{15})</td>
</tr>
<tr>
<td>(10^3)</td>
<td>10</td>
<td>(10^3)</td>
<td>1.0 (\cdot 10^4)</td>
<td>(10^6)</td>
<td>(10^9)</td>
<td>(10^9)</td>
<td>(10^{18})</td>
</tr>
<tr>
<td>(10^4)</td>
<td>13</td>
<td>(10^4)</td>
<td>1.3 (\cdot 10^5)</td>
<td>(10^8)</td>
<td>(10^{12})</td>
<td>(10^{12})</td>
<td>(10^{21})</td>
</tr>
<tr>
<td>(10^5)</td>
<td>17</td>
<td>(10^5)</td>
<td>1.7 (\cdot 10^6)</td>
<td>(10^{10})</td>
<td>(10^{15})</td>
<td>(10^{15})</td>
<td>(10^{24})</td>
</tr>
<tr>
<td>(10^6)</td>
<td>20</td>
<td>(10^6)</td>
<td>2.0 (\cdot 10^7)</td>
<td>(10^{12})</td>
<td>(10^{18})</td>
<td>(10^{18})</td>
<td>(10^{27})</td>
</tr>
</tbody>
</table>

**Table 2.1** Values (some approximate) of several functions important for analysis of algorithms

### Asymptotic Growth Rate

- A way of comparing functions that ignores *constant factors* and *small input sizes*

- \(O(g(n))\): class of functions \(f(n)\) that grow *no faster* than \(g(n)\)

- \(\Theta(g(n))\): class of functions \(f(n)\) that grow *at same rate* as \(g(n)\)

- \(\Omega(g(n))\): class of functions \(f(n)\) that grow *at least as fast* as \(g(n)\)

see figures 2.1, 2.2, 2.3
Figure 2.1 Big-oh notation: \( t(n) \in O(g(n)) \)

Figure 2.2 Big-omega notation: \( t(n) \in \Omega(g(n)) \)
Establishing Order of Growth Using the Definition

Definition: \( f(n) \) is in \( O(g(n)) \) if order of growth of \( f(n) \) ≤ order of growth of \( g(n) \) (within constant multiple), i.e., there exist positive constant \( c \) and non-negative integer \( n_0 \) such that

\[
f(n) \leq c \cdot g(n) \text{ for every } n \geq n_0
\]

Examples:
- \( 10n \) is \( O(n^2) \)
- \( 5n+20 \) is \( O(n) \)

It’s a game of finding the right \( c \) and \( n_0 \)!
Establishing Order of Growth Using the Definition

Definition: \( f(n) \) is in \( \Omega(g(n)) \) if order of growth of \( f(n) \) \( \geq \) order of growth of \( g(n) \) (within constant multiple), i.e., there exist positive constant \( c \) and non-negative integer \( n_0 \) such that

\[
f(n) \geq c \ g(n) \text{ for every } n \geq n_0
\]

Examples:
- \( 10n^2 \) is \( \Omega(n) \)
- \( 5n+20 \) is \( \Omega(n) \)

It’s a game of finding the right \( c \) and \( n_0 \)!

Establishing Order of Growth Using the Definition

- Definition: \( f(n) \) is in \( \Theta(g(n)) \) if order of growth of \( f(n) \) = order of growth of \( g(n) \) (within constant multiple), i.e.,

\[
f(n) \in O(g(n)) \text{ and } f(n) \in \Omega(g(n))
\]

Examples:
- \( 10n^2 \) is \( \Theta(n^2) \)
- \( 5n+20 \) is \( \Theta(n) \)

It’s a game of finding the right \( c \) and \( n_0 \)!
Some Properties of Asymptotic Order of Growth

- \( f(n) \in \Theta(g(n)) \) iff \( f(n) \in O(g(n)) \) and \( f(n) \in \Omega(g(n)) \)
- Theorem: \( f(n) \in O(f(n)) \)
- Theorem: \( f(n) \in O(g(n)) \) iff \( g(n) \in \Omega(f(n)) \)
- Theorem: If \( f(n) \in O(g(n)) \) and \( g(n) \in O(h(n)) \) , then 
  \[ f(n) \in O(h(n)) \]
  (Note similarity with \( a \leq b \))

- Theorem: If \( f_1(n) \in O(g_1(n)) \) and \( f_2(n) \in O(g_2(n)) \) , then 
  \[ f_1(n) + f_2(n) \in O(\max\{g_1(n), g_2(n)\}) \]

Establishing Order of Growth Using Limits

\[
\lim_{n \to \infty} \frac{T(n)}{g(n)} =
\begin{cases} 
0, & \text{order of growth of } T(n) \text{ ___ order of growth of } g(n) \\
c > 0, & \text{order of growth of } T(n) \text{ ___ order of growth of } g(n) \\
\infty, & \text{order of growth of } T(n) \text{ ___ order of growth of } g(n)
\end{cases}
\]

Examples:
- \( 10n \) vs. \( 2n^2 \)
- \( n(n+1)/2 \) vs. \( n^2 \)
- \( \log_b n \) vs. \( \log_c n \)
Establishing Order of Growth Using Limits

\[
\lim_{n \to \infty} \frac{T(n)}{g(n)} = \begin{cases} 
0, & T(n) = O(g(n)) \text{ but not } \Theta(g(n)), \text{ i.e., } o(g(n)) \\
\infty, & T(n) = \Omega(g(n)) \text{ but not } \Theta(g(n)), \text{ i.e., } \omega(g(n)) \\
c > 0, & T(n) = \Theta(g(n)) 
\end{cases}
\]

Examples:

- \(10n\) vs. \(2n^2\)
- \(n(n+1)/2\) vs. \(n^2\)
- \(\log_b n\) vs. \(\log_c n\)

L’Hôpital’s Rule and Stirling’s Formula

L’Hôpital’s rule: If \(\lim_{n \to \infty} f(n) = \lim_{n \to \infty} g(n) = \infty\) and the derivatives \(f', g'\) exist, then

\[
\lim_{n \to \infty} \frac{f(n)}{g(n)} = \lim_{n \to \infty} \frac{f'(n)}{g'(n)}
\]

Example: \(\log n\) vs. \(n\)

Stirling’s formula: \(n! \approx (2\pi n)^{1/2} (n/e)^n\)

Example: \(2^n\) vs. \(n!\)
Orders of Growth of Some Important Functions

- All logarithmic functions \( \log_a n \) belong to the same class \( \Theta(\log n) \) no matter what the logarithm’s base \( a > 1 \) is.

- All polynomials of the same degree \( k \) belong to the same class:
  \[ a_k n^k + a_{k-1} n^{k-1} + \ldots + a_0 \in \Theta(n^k) \]

- Exponential functions \( a^n \) have different orders of growth for different \( a \)'s: order \( a^n < \text{order } b^n \), where \( a < b \)

- order \( \log n < \text{order } n^\alpha (\alpha > 0) < \text{order } a^n \) < order \( n! < \text{order } n^n \)

Basic Asymptotic Efficiency Classes

<table>
<thead>
<tr>
<th>Function</th>
<th>Class</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>constant</td>
</tr>
<tr>
<td>( \log n )</td>
<td>logarithmic</td>
</tr>
<tr>
<td>( n )</td>
<td>linear</td>
</tr>
<tr>
<td>( n \log n )</td>
<td>( n \log n )</td>
</tr>
<tr>
<td>( n^2 )</td>
<td>quadratic</td>
</tr>
<tr>
<td>( n^3 )</td>
<td>cubic</td>
</tr>
<tr>
<td>( 2^n )</td>
<td>exponential</td>
</tr>
<tr>
<td>( n! )</td>
<td>factorial</td>
</tr>
</tbody>
</table>
Time Efficiency of Nonrecursive Algorithms

General Plan for Analysis

- Decide on parameter \( n \) indicating \textit{input size}
- Identify algorithm’s \textit{basic operation}
- Determine \textit{worst}, \textit{average}, and \textit{best} cases for input of size \( n \)
- Set up a sum for the number of times the basic operation is executed
- Simplify the sum using standard formulas and rules (see Appendix A)

Useful Summation Formulas and Rules

\[
\sum_{l \leq i \leq u} 1 = 1 + 1 + \ldots + 1 = u - l + 1
\]

In particular, \( \sum_{l \leq i \leq u} 1 = n - 1 + 1 = n \in \Theta(n) \)

\[
\sum_{1 \leq i \leq n} i = 1 + 2 + \ldots + n = n(n+1)/2 \approx n^2/2 \in \Theta(n^2)
\]

\[
\sum_{1 \leq i \leq n} i^2 = 1^2 + 2^2 + \ldots + n^2 = n(n+1)(2n+1)/6 \approx n^3/3 \in \Theta(n^3)
\]

\[
\sum_{0 \leq i \leq n} a^i = 1 + a + \ldots + a^n = (a^{n+1} - 1)/(a - 1) \text{ for any } a \neq 1
\]

In particular, \( \sum_{0 \leq i \leq n} 2^i = 2^0 + 2^1 + \ldots + 2^n = 2^{n+1} - 1 \in \Theta(2^n) \)

\[
\Sigma(a_i \pm b_i) = \Sigma a_i \pm \Sigma b_i \quad \Sigma ca_i = c\Sigma a_i \quad \Sigma_{l \leq i \leq u} a_i = \Sigma_{l \leq i \leq m} a_i + \Sigma_{m+1 \leq i \leq u} a_i
\]
Examples

- Maximum element
- Element uniqueness problem
- Matrix multiplication
- Counting binary digits
- Mystery Algorithm

Example 1: Maximum Element

**Algorithm**  \( MaxElement(A[0..n - 1]) \)

// Determines the value of the largest element in a given array
// Input: An array \( A[0..n - 1] \) of real numbers
// Output: The value of the largest element in \( A \)

\[ \text{maxval} \leftarrow A[0] \]

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

\[ \text{if } A[i] > \text{maxval} \]

\[ \text{maxval} \leftarrow A[i] \]

\[ \text{return } \text{maxval} \]
Example 2: Element Uniqueness Problem

```
ALGORITHM UniqueElements(A[0..n−1])
  // Determines whether all the elements in a given array are distinct
  // Input: An array A[0..n−1]
  // Output: Returns “true” if all the elements in A are distinct
  //         and “false” otherwise
  for i ← 0 to n − 2 do
    for j ← i + 1 to n − 1 do
  return true
```

Example 3: Matrix Multiplication

```
ALGORITHM MatrixMultiplication(A[0..n−1, 0..n−1], B[0..n−1, 0..n−1])
  // Multiplies two n-by-n matrices by the definition-based algorithm
  // Input: Two n-by-n matrices A and B
  // Output: Matrix C = AB
  for i ← 0 to n − 1 do
    for j ← 0 to n − 1 do
      C[i, j] ← 0.0
      for k ← 0 to n − 1 do
  return C
```
Example 4: Counting Binary Digits

**ALGORITHM** \textit{Binary}(n)

//Input: A positive decimal integer \(n\)
//Output: The number of binary digits in \(n\)’s binary representation

\[
\text{count} \leftarrow 1 \\
\text{while } n > 1 \text{ do} \\
\hspace{1em} \text{count} \leftarrow \text{count} + 1 \\
\hspace{1em} n \leftarrow \lfloor n/2 \rfloor \\
\text{return count}
\]

It cannot be investigated the way the previous examples are. (Why?)

Example 5: Mystery Algorithm

\[
\text{for } i := 1 \text{ to } n - 1 \text{ do} \\
\hspace{1em} \text{max} := i \\
\hspace{1em} \text{for } j := i + 1 \text{ to } n \text{ do} \\
\hspace{2em} \text{if } |A[j, i]| > |A[\text{max}, i]| \text{ then max := } j \\
\hspace{1em} \text{for } k := i \text{ to } n + 1 \text{ do} \\
\hspace{2em} \text{swap } A[i, k] \text{ with } A[\text{max}, k] \\
\hspace{1em} \text{for } j := i + 1 \text{ to } n \text{ do} \\
\hspace{2em} \text{for } k := n + 1 \text{ downto } i \text{ do} \\
\]
Plan for Analysis of Recursive Algorithms

- Decide on a parameter indicating an input’s size.
- Identify the algorithm’s basic operation.
- Check whether the number of times the basic op. is executed may vary on different inputs of the same size. (If it may, the worst, average, and best cases must be investigated separately.)
- Set up a recurrence relation with an appropriate initial condition expressing the number of times the basic op. is executed.
- Solve the recurrence (or, at the very least, establish its solution’s order of growth) by backward substitutions or another method.

Example 1: Recursive Evaluation of $n!$

Definition: $n! = 1 \times 2 \times \ldots \times (n-1) \times n$ for $n \geq 1$ and $0! = 1$

Recursive definition of $n!$: $F(n) = F(n-1) \times n$ for $n \geq 1$ and $F(0) = 1$

ALGORITHM $F(n)$

//Computes $n!$ recursively
//Input: A nonnegative integer $n$
//Output: The value of $n!$
if $n = 0$ return 1
else return $F(n-1) \times n$

Trace the algorithm to set up recurrence!
Solving the Recurrence for $M(n)$

$M(n) = M(n-1) + 1, \ M(0) = 0$

Example 2: The Tower of Hanoi Puzzle

Recurrence for number of moves:
Solving Recurrence for Number of Moves

\[ M(n) = 2M(n-1) + 1, \quad M(1) = 1 \]
Example 3: Counting #Bits

**ALGORITHM**  
*BinRec(n)*  
//Input: A positive decimal integer *n*  
//Output: The number of binary digits in *n*'s binary representation  
if *n* = 1 return 1  
else return *BinRec([n/2]) + 1*

**Important Recurrence Types:**

- One (constant) operation reduces problem size by one.  
  \[ T(n) = T(n-1) + c \quad T(1) = d \]  
  Solution: \[ T(n) = (n-1)c + d \] **linear**

- A pass through input reduces problem size by one.  
  \[ T(n) = T(n-1) + cn \quad T(1) = d \]  
  Solution: \[ T(n) = \lfloor n(n+1)/2 - 1 \rfloor c + d \] **quadratic**

- One (constant) operation reduces problem size by half.  
  \[ T(n) = T(n/2) + c \quad T(1) = d \]  
  Solution: \[ T(n) = c \log n + d \] **logarithmic**

- A pass through input reduces problem size by half, and both halves are needed.  
  \[ T(n) = 2T(n/2) + cn \quad T(1) = d \]  
  Solution: \[ T(n) = cn \log n + d n \] **n log n**
Fibonacci Numbers

- The Fibonacci sequence:
  0, 1, 1, 2, 3, 5, 8, 13, 21, ...

- Fibonacci recurrence:
  \[ F(n) = F(n-1) + F(n-2) \]
  \[ F(0) = 0 \]
  \[ F(1) = 1 \]

- General 2\textsuperscript{nd} order linear homogeneous recurrence with constant coefficients (LHRCCs):
  \[ aX(n) + bX(n-1) + cX(n-2) = 0 \]

- Another example:
  \[ A(n) = 3A(n-1) - 2A(n-2) \quad A(0) = 1 \quad A(1) = 3 \]

Solving \( aX(n) + bX(n-1) + cX(n-2) = 0 \)

- Set up the characteristic equation (quadratic)
  \[ ar^2 + br + c = 0 \]

- Solve to obtain roots \( r_1 \) and \( r_2 \)

- General solution to the recurrence
  if \( r_1 \) and \( r_2 \) are two distinct real roots:
    \[ X(n) = \alpha r_1^n + \beta r_2^n \]
  if \( r_1 = r_2 = r \) are two equal real roots:
    \[ X(n) = \alpha r^n + \beta nr^n \]

- Particular solution can be found by using initial conditions
**Example of Solving 2nd Order LHRCCs**

- \( A(n) = 3A(n-1) - 2(n-2) \), where \( A(0) = 1 \quad A(1) = 3 \)

- Characteristic equation (quadratic):

- Solve to obtain roots \( r_1 \) and \( r_2 \):

- General solution to RR: linear combination of \( r_1^n \) and \( r_2^n \)

- Particular solution using initial conditions:

**Application to the Fibonacci Numbers**

\[ F(n) = F(n-1) + F(n-2) \quad \text{or} \quad F(n) - F(n-1) - F(n-2) = 0 \]

Characteristic equation:

Roots of the characteristic equation:

General solution to the recurrence:

Particular solution for \( F(0) = 0, F(1)=1 \):
Computing Fibonacci Numbers

1. Definition based recursive algorithm

2. Nonrecursive definition-based algorithm

3. Explicit formula algorithm

\[ F(n) = \frac{1}{\sqrt{5}} \phi^n \text{ round to the nearest integer, where } \phi = \frac{1+\sqrt{5}}{2}, \text{ the golden ratio} \]

4. Logarithmic algorithm based on formula:

\[
\begin{pmatrix}
F(n-1) & F(n) \\
F(n) & F(n+1)
\end{pmatrix} =
\begin{pmatrix}
0 & 1 \\
1 & 1
\end{pmatrix}^n
\]

for \( n \geq 1 \), assuming an efficient way of computing matrix powers.