Algorithm Analysis == Complexity Analysis

Program Running (Execution) Time Factors
- Machine Speed
- Programming Language
- Compiler Code Generation
- Input Data Size
- Time Complexity of Algorithm
  † Number of executed statements
  † Function of the size of the input (termed n)

Running Time Factor Implications
- Compiler code generation & processor speed differences are too great to be used as a basis for impartial algorithm comparisons.

- Running Time Calculation Rules
  - Running of assignment & I/O statements take time T(1).
    † Unit of time is arbitrary
  - Running time of a sequence of statements is the largest time of any statement in the sequence.
  - Running time of an IF statement is the condition evaluation T(1) + MAX( statements executed when true or false ).
  - Loop execution time is the sum, over the number of times the loop is executed, of the body time + T(1) for the loop setup and overhead, (e.g., while condition check, for initialization, check & increment).
  † Always assume that the loop executes the maximum number of iterations possible
Order of Magnitude

Function Estimation
- Given an algorithm that takes time:
  \[ T(n) = 3n^2 + 5n + 100 \]
  \( T(n) \approx O(n^2) \) for large values of \( n \), thus
  Order is used to compare complexity.

- \( n^2 \) forms an “upper bound” on \( T(n) \) (asymptotic bound)

Big-O Notation
- Formally:
  \[ f(n) \text{ is } O(g(n)) \text{ if there are constants } c > 0 \text{ and } N > 0, \text{ for which } f(n) \leq cg(n) \text{ for all } n > N. \]
- Shortcuts to determine \( O(f(n)) \):
  \[ O(f(n)) = O(\text{dominant term of } f(n)) \]
  \[ O(\text{dominant term of } f(n)) = O(\text{dominant term of } f(n) \text{ omitting any coefficients of the dominant term}) \]
  \[ O(\text{constants}) = O(1) \]
  \[ O(\text{omitting the bases of the logarithms}) \]

Simple Summation

Example 1
- Given:
  \[
  \begin{align*}
  &\text{for } (i = 0; i < n-1; i++) \\
  &\text{for } (j = 0; j < i; j++) \\
  &\text{array}[i][j] = 0;
  \end{align*}
  \]

- Analysis:
  \[ \uparrow \text{ Loop } i \text{ will execute } n-1 \text{ times -- this is the external sum.} \]
  \[ \uparrow \text{ Loop } j \text{ will execute } i \text{ times -- this is the internal sum.} \]
  \[ \uparrow \text{ The assignment statement is } T(1). \]

\[
\sum_{i=1}^{n-1} \sum_{j=1}^{i} 3 = 3 \sum_{i=1}^{n-1} i = 3 \sum_{i=1}^{n-1} (n-i)
\]

- Analysis of function:
  \[ = 3 (1 + 2 + \ldots + n-2 + n-1) \]
  \[ = 3 \frac{2}{2} (1 + 2 + \ldots + n-2 + n-1) \]
  \[ = 3 \frac{1}{2} (1 + 1 + 2 + 2 + \ldots + n-1 + n-1) \]
  \[ = 3 \frac{1}{2} ((1+n-1) + (2+n-2) + \ldots + (n-1+1)) \]
  \[ = 3 \frac{1}{2} (n + n + \ldots + n) \]
  \[ = 3 \frac{1}{2} ((n - 1) n) \]
  \[ = 3 \frac{n^2 - n}{2} \]
  \[ O(3 \frac{n^2 - n}{2}) = O(3n^2/2 - 3n/2) = O(n^2) \]

For ease of summation count loop repetitions starting at 1.
Array Summation: Order Analysis

Order of Magnitude Analysis (Big O)

- Order from a..h
  - a: $T(1)$ by rule 1
  - b: Sum from 0..n over statements c..h by rule 4
  - c: $O(T(1)) = T(1)$ by rules 1 & 2
  - d: Sum from 0..i over statements e..f by rule 4
  - e,f,g,h: $T(1)$ by rule 1
  - ef: $T(1)$ by rule 2

- Analysis will deal with statements (a .. h)

- $d = O(\sum_{j=1}^{i+1} j) = O(i+1) = O(i)$

- $cdgh = O(i) = \max(O(1), O(i), O(1), O(1))$ by rule 2

- $b = O\left(\sum_{i=1}^{n+1} i\right) = O\left(\frac{(n+1)(n+2)}{2}\right) = O\left(n^2\right)$

- $O(a, b) = \max(O(1), O(n)) = O(n)$

Array Summation: Time Analysis

Running Time Analysis

- With constants of proportionality
- Note: while, if conditions count as 1

- $= a + b\left(c + d(e + f) + g + h\right)$

- $= 1 + \sum_{i=1}^{n+1} \left(3 + \sum_{j=1}^{i+1} 3 + 2\right)$

- $= 1 + \sum_{i=1}^{n+1} (8) + 3 \sum_{i=1}^{n+1} (i)$

- $= 1 + 8(n+1) + 3\left(\frac{n+1}{2}\right)\left(\frac{n+2}{2}\right)$

- $= \frac{3}{2}n^2 + \frac{25}{2}n + 12$
Common Complexity Classes (growth curves)

Observations
- **constants of proportionality**, (coefficients & lesser terms), have very little effect for large values of n.
- For small problems the complexity makes little difference
- Large problems with **Order > nlog(n)** cannot practically be executed
  † For n = 1000 (medium problems) $n^2$ algorithms can still be used

Practical Applications

Assume:
- 1 day = 100,000 sec. = $10^5$
- Input size $n = 10^6$
- A computer that executes 1,000,000 Inst/sec
  † C statement instructions

Algorithm Complexity Class Comparison

<table>
<thead>
<tr>
<th>Order: $n^2$</th>
<th>Order: $n \log_2 n$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$(10^6)^2$ Inst</td>
<td>$10^6 \log_2 10^6$ Inst</td>
</tr>
<tr>
<td>$10^{12}$ Inst</td>
<td>$20 (10^6)$</td>
</tr>
<tr>
<td>$10^{12} / 10^6$ secs to run</td>
<td>$2 (10^7)$</td>
</tr>
<tr>
<td>$10^6$ secs to run</td>
<td>$10^6 / 10^6$ run secs</td>
</tr>
<tr>
<td>$10^6$ / $10^5$ days to run</td>
<td>$20$ sec to run</td>
</tr>
</tbody>
</table>

Internal Class Comparisons
- Within complexity classes the differences between algorithms due to **constants of proportionality**, (coefficients & lesser terms), are not significant enough to warrant reporting except for certain (high usage) applications (e.g., sorting, searching)
Recursion Analysis

Recurrence Relations
- Mathematical functions that define the running time of recursive functions.
- Recurrence relation functions are defined recursively in terms of the recurrence relation itself.

Method
- Step 1:
  Determine T(n) for the general case, treating recursive calls as T(?).
- Step 2:
  Determine T(0) or T(1), etc. (i.e. the base cases).
- Step 3: (unrolling aka iterative expansion)
  Expand T(n) determined in step 1 for T(n-1), T(n-2), etc.
- Step 4: (pattern search)
  Examine the expanded formulas, collect terms, reduce algebraically.
- Step 5:
  Apply an appropriate summation formula to the reduced formula.
- Step 6:
  Solve (reduce) the summation to determine:
  T(n) = polynomial
- Step 7:
  Apply Big-O to determine the order of O(T(n)).

Analysis: Recursive Factorial

Recursive Factorial Function
```
int factorial ( int n ) {
    if ( n <= 1 )                        // A
        return ( 1 );                     // B
    else
        return( n * factorial( n-1 ) );   // C
}
```

Analysis
- Statement A: takes time T(1) for the condition evaluation
- Statement B: takes time T(1) for the return assignment
- Statement C: takes time T(1) for the other operations
  (multiplication & return) + T(n-1), the time to determine !n(1)
Analysis: Recursive Factorial

Analysis Continued

T(n) = T(condition evaluation) + T(other operations) + T(!n-1))
   = 1 + 1 + T(n-1)
   = T(n-1) + C

T(0) = T(1+1) = T(2) = K (condition + return assignment)
T(1) = T(1+1) = T(2) = K (condition + return assignment)
T(n-1) = T(n-2) + C substituting (n-1) in place of n above.
T(n) = T(n-2) + C + C expanding T(n) above

When i = n-1:
T(n) = K + (n-1) C
O(T(n)) = O(K + (n-1) C)
   = O(n)

When i = n:
T(n) = K + nC
O(T(n)) = O(K + nC)
   = O(n)

Analysis: Recursive Summation

Recursive Summation Function

```c
int rSum ( const float ray[], int n )
{
    //recursive array summation
    if ( n <= 0 ) // A
        return 0; // B
    else
        return(rSum(ray, n-1) + ray[n]); // C
}
```

Analysis

- Statement A takes time T(1) for the condition evaluation
- Statement B takes time T(1) for the return assignment
- Statement C takes time T(1) for the other operations (addition & return) + T(n-1), the time to determine rsum(ray, n-1)
Analysis: Recursive Summation

Analysis Continued

\[ T(n) = T(\text{condition evaluation}) + T(\text{addition}) + T(\text{rsum(ray,n-1)}) \]
\[ = 1 + 1 + T(n-1) \]
\[ = T(n-1) + C \quad \text{(constant for time perform condition evaluation and other operations)} \]

\[ T(0) = T(1) + 1 = T(2) = K \quad \text{(condition + return assignment)} \]
\[ T(n-1) = T(n-2) + C \quad \text{substituting (n-1) in place of n above.} \]
\[ T(n) = T(n-2) + C + C \quad \text{expanding T(n) above} \]
\[ T(n) = T(n-3) + C + C + C \]

\[ \vdots \]
\[ T(n) = T(1) + C + C + \ldots + C = K + C + C + \ldots + C \]
\[ = T(n-i) + iC, \text{ for } n > i \]

When \( i = n-1: \)
\[ T(n) = K + (n-1) C \]
\[ O(T(n)) = O(K + (n-1) C) \]
\[ = O(n) \]

When \( i = n: \)
\[ T(n) = K + (n)C \]
\[ O(T(n)) = O(K + (n) C) \]
\[ = O(n) \]

Algorithm Behavior

Categories

- Algorithms must be examined under different situations to correctly determine their efficiency for accurate comparisons.

Best Case Analysis

- Assumes the input, data etc. are arranged in the most advantageous order for the algorithm, i.e. causes the execution of the fewest number of instructions.

  - E.g., sorting - list is already sorted; searching - desired item is located at first accessed position.

Worst Case Analysis

- Assumes the input, data etc. are arranged in the most disadvantageous order for the algorithm, i.e. causes the execution of the largest number of statements.

  - E.g., sorting - list is in opposite order; searching - desired item is located at the last accessed position or is missing.

Average Case Analysis

- Determines the average of the running times over all possible permutations of the input data.

  - E.g., searching - desired item is located at every position, for each search), or is missing.

Caveats

- Algorithms may have quite different Orders for the analysis categories, e.g., \( O(1) \), \( O(n^2) \), \( O(n\log n) \), respectively.