Big-O Performance Analysis

Execution Time Factors

• Computer:
  – CPU speed, amount of memory, etc.
• Compiler:
  – Efficiency of code generation.
• Data:
  – Number of items to be processed.
  – Initial ordering (e.g., random, sorted, reversed)
• Algorithm:
  – E.g., linear vs. binary search.
Are Algorithms Important?

- The fastest algorithm for 100 items may not be the fastest for 10,000 items!
- Algorithm choice is more important than any other factor!

Counting the instructions

```java
public void SelectionSort ( int [] num ){
    int i, j, first, temp;
    for ( i = num.length - 1; i > 0; i -- )
    {
        first = 0;  //Initialize to subscript of first element
        for(j = 1; j <= i; j ++)
        {
            if( num[j] < num[first] )
                first = j;
        }
        temp = num[first];  //swap smallest found with element in position i.
        num[first] = num[i];
        num[i] = temp;
    }
}
```

\[4 \times (n-1) + 2 \times (n-2) + 4 \times (n-2) + 2 \times (n-2) + 4 + 2^1 =
4(n-1) + 2((n-1)+(n-2)+(n-3)+\ldots)+1 = 4(n-1) * 2 \times n(n-1)/2
= 4(n-1) + n^2 - n = n^2 + 3n - 4\]
What is Big-O?

- Big-O characterizes algorithm performance.
- Big-O describes how execution time grows as the number of data items increase.
- Big-O is a function with parameter $N$, where $N$ represents the number of items.

Predicting Execution Time

- If a program takes 10ms to process one item, how long will it take for 1000 items?

\[
\text{(time for 1 item) } \times \left( \text{Big-O( ) time complexity of } N \text{ items} \right)
\]

<table>
<thead>
<tr>
<th>Complexity</th>
<th>Time for 1000 Items</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\log_{10} N$</td>
<td>3 x 10ms, 0.03 sec</td>
</tr>
<tr>
<td>$N$</td>
<td>$10^3 \times 10$ms, 10 sec</td>
</tr>
<tr>
<td>$N \log_{10} N$</td>
<td>$10^3 \times 3 \times 10$ms, 30 sec</td>
</tr>
<tr>
<td>$N^2$</td>
<td>$10^6 \times 10$ms, 16 min</td>
</tr>
<tr>
<td>$N^3$</td>
<td>$10^9 \times 10$ms, 12 days</td>
</tr>
</tbody>
</table>
Complexity

• In general, we are not so much interested in the time and space complexity for small inputs.

• For example, while the difference in time complexity between linear and binary search is meaningless for a sequence with \( n = 10 \), it is gigantic for \( n = 2^{30} \).

Complexity

• For example, let us assume two algorithms A and B that solve the same class of problems.

• The time complexity of A is \( 5000n \), the one for B is \( [1.1^n] \) for an input with \( n \) elements.

• For \( n = 10 \), A requires 50,000 steps, but B only 3, so B seems to be superior to A.

• For \( n = 1000 \), however, A requires 5,000,000 steps, while B requires \( 2.5 \cdot 10^{41} \) steps.
Complexity

• This means that algorithm B cannot be used for large inputs, while algorithm A is still feasible.

• So what is important is the **growth** of the complexity functions.

• The growth of time and space complexity with increasing input size n is a suitable measure for the comparison of algorithms.

### Complexity

**Comparison: time complexity of algorithms A and B**

<table>
<thead>
<tr>
<th>Input Size</th>
<th>Algorithm A</th>
<th>Algorithm B</th>
</tr>
</thead>
<tbody>
<tr>
<td>n</td>
<td>5,000n</td>
<td>$1.1^n$</td>
</tr>
<tr>
<td>10</td>
<td>50,000</td>
<td>3</td>
</tr>
<tr>
<td>100</td>
<td>500,000</td>
<td>13,781</td>
</tr>
<tr>
<td>1,000</td>
<td>5,000,000</td>
<td>$2.5 \cdot 10^{41}$</td>
</tr>
<tr>
<td>1,000,000</td>
<td>$5 \cdot 10^9$</td>
<td>$4.8 \cdot 10^{1392}$</td>
</tr>
</tbody>
</table>
The Growth of Functions

• The growth of functions is usually described using the **big-O notation**.

• **Definition:** Let $f$ and $g$ be functions from the integers or the real numbers to the real numbers.
• We say that $f(x)$ is $O(g(x))$ if there are constants $C$ and $k$ such that

• $|f(x)| \leq C|g(x)|$
• whenever $x > k$.

The Growth of Functions

• When we analyze the growth of *complexity functions*, $f(x)$ and $g(x)$ are always positive.

• Therefore, we can simplify the big-O requirement to

• $f(x) \leq C \cdot g(x)$ whenever $x > k$.

• If we want to show that $f(x)$ is $O(g(x))$, we only need to find **one** pair $(C, k)$ (which is never unique).
The Growth of Functions

• The idea behind the big-O notation is to establish an upper boundary for the growth of a function \( f(x) \) for large \( x \).
• This boundary is specified by a function \( g(x) \) that is usually much simpler than \( f(x) \).
• We accept the constant \( C \) in the requirement
  • \( f(x) \leq C \cdot g(x) \) whenever \( x > k \),
  • because \( C \) does not grow with \( x \).
• We are only interested in large \( x \), so it is OK if \( f(x) \geq C \cdot g(x) \) for \( x \leq k \).

What is Big-O

\[ f(n) = O(g(n)) \text{ iff } \exists \text{ positive constants } c \text{ and } n_0 \text{ such that } 0 \leq f(n) \leq cg(n) \forall n \geq n_0. \]
Big-O Example

\[ f(x) = 6x^4 - 2x^3 + 5 \]

Prove \( f(x) = \mathcal{O}(n^4) \)

\[
|6x^4 - 2x^3 + 5| \leq 6x^4 + |2x^3| + 5 \\
\leq 6x^4 + 2x^4 + 5x^4 \\
= 13x^4
\]

The Growth of Functions

• Example:
  • Show that \( f(x) = x^2 + 2x + 1 \) is \( \mathcal{O}(x^2) \).
  
  • For \( x > 1 \) we have:
    • \( x^2 + 2x + 1 \leq x^2 + 2x^2 + x^2 \)
    • \( \Rightarrow x^2 + 2x + 1 \leq 4x^2 \)
  
  • Therefore, for \( C = 4 \) and \( k = 1 \):
    • \( f(x) \leq Cx^2 \) whenever \( x > k \).
    
    • \( \Rightarrow f(x) \) is \( \mathcal{O}(x^2) \).
# Common Growth Rates

<table>
<thead>
<tr>
<th>Big-O Characterization</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>$O(1)$</td>
<td>constant</td>
</tr>
<tr>
<td>$O(\log N)$</td>
<td>log</td>
</tr>
<tr>
<td>$O(N)$</td>
<td>linear</td>
</tr>
<tr>
<td>$O(N \log N)$</td>
<td>$n$-log-$n$</td>
</tr>
<tr>
<td>$O(N^2)$</td>
<td>quadratic</td>
</tr>
<tr>
<td>$O(N^3)$</td>
<td>cubic</td>
</tr>
<tr>
<td>$O(2^N)$</td>
<td>exponential</td>
</tr>
</tbody>
</table>

- Adding to the front of a linked list
- Binary search
- Linear search
- Binary merge sort
- Bubble Sort
- Simultaneous linear equations
- The Towers of Hanoi problem
The Growth of Functions

Question: If \( f(x) \) is \( O(x^2) \), is it also \( O(x^3) \)?

Yes. \( x^3 \) grows faster than \( x^2 \), so \( x^3 \) grows also faster than \( f(x) \).

Therefore, we always have to find the smallest simple function \( g(x) \) for which \( f(x) \) is \( O(g(x)) \).

The Growth of Functions

“Popular” functions \( g(n) \) are

\( n \log n, 1, 2^n, n^2, n!, n, n^3, \log n \)

Listed from slowest to fastest growth:

\( 1 \)
\( \log n \)
\( n \)
\( n \log n \)
\( n^2 \)
\( n^3 \)
\( 2^n \)
\( n! \)
The Growth of Functions

• A problem that can be solved with polynomial worst-case complexity is called **tractable**.

• Problems of higher complexity are called **intractable**.

• Problems that no algorithm can solve are called **unsolvable**.

Determining Big-O: Repetition

```plaintext
for (i = 1; i <= n; i++)
{  
m = m + 2 ;  // constant time
}
```

Total time = (a constant c) * n = cn = O(N)

*Ignore multiplicative constants (e.g., “c”).*
Determining Big-O: Repetition

```
for (i = 1; i <= n; i++)
{
    for (j = 1; j <= n; j++)
    {
        k = k+1 ;
    }
}
```

constant time

**outer loop** executed *n* times

**inner loop** executed *n* times

Total time = \(c \times n \times n \times = cn^2 = O(N^2)\)

---

Determining Big-O: Repetition

```
for (i = 1; i <= n; i++)
{
    for (j = 1; j <= 100; j++)
    {
        k = k+1 ;
    }
}
```

constant time

**outer loop** executed *n* times

**inner loop** executed *100* times

Total time = \(c \times 100 \times n \times = 100cn = O(N)\)
Determining Big-O: Sequence

\[
\begin{align*}
\text{constant time (} c_0 \text{)} & \quad x = x + 1; \\
\text{constant time (} c_1 \text{)} & \quad \text{for (i=1; i<=n; i++)} \\
& \quad \{ \\
& \quad \quad m = m + 2; \\
& \quad \} \\
\text{outer loop} & \quad \text{executed } n \text{ times} \\
\text{inner loop} & \quad \text{executed } n \text{ times} \\
\text{constant time (} c_2 \text{)} & \quad \text{for (i=1; i<=n; i++)} \\
& \quad \{ \\
& \quad \quad \text{for (j=1; j<=n; j++)} \\
& \quad \quad \{ \\
& \quad \quad \quad k = k + 1; \\
& \quad \quad \} \\
& \quad \} \\
\end{align*}
\]

Total time = \( c_0 + c_1 n + c_2 n^2 \) = \( O(N^2) \)

Only dominant term is used

---

Determining Big-O: Selection

\[
\begin{align*}
\text{test: constant (} c_0 \text{)} & \quad \text{if (depth() != otherStack.depth())} \\
& \quad \{ \\
& \quad \quad \text{return false; } \\
& \quad \} \\
\text{else} & \quad \text{then part: constant (} c_1 \text{)} \\
& \quad \{ \\
& \quad \quad \text{for (int n = 0; n < depth(); n++)} \\
& \quad \quad \{ \\
& \quad \quad \quad \text{return false; } \\
& \quad \quad \} \\
& \quad \} \\
\text{another if: constant (} c_2 \text{)} & \quad \text{if (!list[n].equals(otherStack.list[n]))} \\
& \quad \{ \\
& \quad \quad \text{return false; } \\
& \quad \} \\
\text{then (} c_1 \text{)} & \quad \text{else part: (} c_2 + c_3 \text{) * n} \\
\end{align*}
\]

Total time = \( c_0 + \text{Worst-Case}(c_1, c_2 + c_3) \) * n = \( O(N) \)

Total time = \( c_0 + \text{Worst-Case}(\text{then, else}) \)

Total time = \( c_0 + \text{Worst-Case}(c_1, \text{else}) \)