Introduction to Analysis of Algorithms
Objectives

• To introduce the concept of analysing algorithms with respect to the time taken to have them executed
  • Purpose:
    • To see if an algorithm is practical
    • To compare different algorithms for solving a problem
Introduction to Analysis of Algorithms

• One aspect of software quality is the efficient use of computer resources:
  • CPU time
  • Memory usage
• We frequently want to analyse algorithms with respect to execution time
  • Called time complexity analysis
  • For example, to decide which sorting algorithm will take less time to run
Time Complexity

• Analysis of time taken is based on:
  • Problem size (e.g. number of items to sort)
  • Primitive operations (e.g. comparison of two values)
• What we want to analyse is the relationship between
  • The size of the problem, $n$
  • And the time it takes to solve the problem, $t(n)$
    • Note that $t(n)$ is a function of $n$, so it depends on the size of the problem
Time Complexity Functions

• This $t(n)$ is called a time complexity function

• What does a time complexity function look like?
  • Example of a time complexity function for some algorithm:
    
    $t(n) = 15n^2 + 45n$

    • See the next slide to see how $t(n)$ changes as $n$ gets bigger!
Example: $15n^2 + 45n$

<table>
<thead>
<tr>
<th>No. of items $n$</th>
<th>$15n^2$</th>
<th>$45n$</th>
<th>$15n^2 + 45n$</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>15</td>
<td>45</td>
<td>60</td>
</tr>
<tr>
<td>2</td>
<td>60</td>
<td>90</td>
<td>150</td>
</tr>
<tr>
<td>5</td>
<td>375</td>
<td>225</td>
<td>600</td>
</tr>
<tr>
<td>10</td>
<td>1,500</td>
<td>450</td>
<td>1,950</td>
</tr>
<tr>
<td>100</td>
<td>150,000</td>
<td>4,500</td>
<td>154,500</td>
</tr>
<tr>
<td>1,000</td>
<td>15,000,000</td>
<td>45,000</td>
<td>15,045,000</td>
</tr>
<tr>
<td>10,000</td>
<td>1,500,000,000</td>
<td>450,000</td>
<td>1,500,450,000</td>
</tr>
<tr>
<td>100,000</td>
<td>150,000,000,000</td>
<td>4,500,000</td>
<td>150,004,500,000</td>
</tr>
<tr>
<td>1,000,000</td>
<td>15,000,000,000,000</td>
<td>45,000,000</td>
<td>15,000,045,000,000</td>
</tr>
</tbody>
</table>
Comparison of Terms in $15n^2 + 45n$

- When $n$ is small, which term is larger?
- But, as $n$ gets larger, note that the $15n^2$ term grows more quickly than the $45n$ term.
- We say that the $n^2$ term is *dominant* in this expression.
Big-Oh Notation

• We require a measurement of the time complexity of an algorithm that is *independent* on any implementation details (programming language and computer that will execute the algorithm).

• i.e. we generally only care about number of operations performed, not how long it takes since that varies from one machine to another.
Big-Oh Notation

• The key issue is the *asymptotic complexity* of the function or *how it grows as n increases*
  • This is determined by the *dominant term* in the growth function (the term that increases most quickly as n increases)
  • Constants become irrelevant as n increases since we want a characterization of the time complexity of an algorithm that is *independent* of the computer that will be used to execute it. Since different computers differ in speed by a constant factor, constant factors are ignored when expressing the asymptotic complexity of a function.
Big-Oh Notation

• The asymptotic complexity of the function is referred to as the \textit{order of} the function, and is specified by using \textit{Big-Oh notation}.
  • \textit{Example}: $O(n^2)$ means that the time taken by the algorithm grows like the $n^2$ function as $n$ increases
  • $O(1)$ means constant time, regardless of the size of the problem
# Some Growth Functions and Their Asymptotic Complexities

<table>
<thead>
<tr>
<th>Growth Function</th>
<th>Order</th>
</tr>
</thead>
<tbody>
<tr>
<td>$t(n) = 17$</td>
<td>$O(1)$</td>
</tr>
<tr>
<td>$t(n) = 20n - 4$</td>
<td>$O(n)$</td>
</tr>
<tr>
<td>$t(n) = 12n \cdot \log_2 n + 100n$</td>
<td>$O(n \cdot \log_2 n)$</td>
</tr>
<tr>
<td>$t(n) = 3n^2 + 5n - 2$</td>
<td>$O(n^2)$</td>
</tr>
<tr>
<td>$t(n) = 2^n + 18n^2 + 3n$</td>
<td>$O(2^n)$</td>
</tr>
</tbody>
</table>
Comparison of Some Typical Growth Functions

- $t(n) = n^3$
- $t(n) = n^2$
- $t(n) = n \log_2 n$
- $t(n) = n$
# Exercise: Asymptotic Complexities

<table>
<thead>
<tr>
<th>Growth Function</th>
<th>Order</th>
</tr>
</thead>
<tbody>
<tr>
<td>( t(n) = 5n^2 + 3n )</td>
<td>?</td>
</tr>
<tr>
<td>( t(n) = n^3 + \log_2 n - 4 )</td>
<td>?</td>
</tr>
<tr>
<td>( t(n) = \log_2 n \times 10n + 5 )</td>
<td>?</td>
</tr>
<tr>
<td>( t(n) = 3n^2 + 3n^3 + 3 )</td>
<td>?</td>
</tr>
<tr>
<td>( t(n) = 2^n + 18n^{100} )</td>
<td>?</td>
</tr>
</tbody>
</table>
Determining Time Complexity

• Algorithms frequently contain sections of code that are executed over and over again, i.e. **loops**

• Analysing loop execution is vital in determining time complexity.
Analysing Loop Execution

• A loop executes a certain number of times (say $n$), so the time complexity of the loop is $n$ times the time complexity of the body of the loop

• **Example**: what is the time complexity of the following loop, in Big-O notation?

```plaintext
x = 0;
for (int i=0; i<n; i++)
    x = x + 1;
```
• **Nested loops**: the body of the outer loop includes the inner loop

• **Example**: what is the time complexity of the following loop, in Big-O notation? Read the next set of notes from the course’s webpage to see how the time complexity of this algorithm and the algorithms in the following pages are computed.

```c
for (int i=0; i<n; i++) {
    x = x + 1;
    for (int j=0; j<n; j++)
        y = y - 1;
}
```
More Loop Analysis Examples

```c
x = 0;
for (int i=0; i<n; i=i+2) {
    x = x + 1;
}

x = 0;
for (int i=1; i<n; i=i*2) {
    x = x + 1;
}
```
More Loop Analysis Examples

```c
x = 0;
for (int i=0; i<n; i++)
    for (int j = i; j < n; j ++)
    {
        x = x + 1;
    }
```
Analysis of Stack Operations

• Stack operations are generally efficient, because they all work on only one end of the collection

• But which is more efficient: the array implementation or the linked list implementation?
Analysis of Stack Operations

- $n$ is the number of items on the stack
- **push** operation for **ArrayStack**:
  - $O(1)$ if array is not full (why?)
  - What would it be if the array is full? *(worst case)*
- **push** operation for **LinkedStack**:
  - $O(1)$ (why?)
- **pop** operation for each?
- **peek** operation for each?