COMP 2140 - Data Structures

Shahin Kamali

Topic 3 - Algorithm Analysis

University of Manitoba

Based on notes by S. Durocher.
Overview

- time complexity
- measuring the time complexity of an algorithm
- Why Big-Oh?
- intuitively understanding Big-Oh
- formally defining Big-Oh
Sequential Search

We will analyze the running time of the following Java implementation of a sequential search algorithm.

```java
// precondition: array A and key are initialized
// postcondition: return i such that A[i] == key
// or i = length of A if for all i, A[i] != key

public int search(int[] A, int key) {
    int n = A.length;
    int i = 0;
    while (i < n) {
        if (A[i] == key) {
            break;
        }
        i++;
    }
    return i;
}
```
Algorithm Analysis

Definition

**Algorithm analysis** refers to the process of deriving estimates of the resource requirements or efficiency of an algorithm.

possible measures of efficiency:

- time taken
- amount of storage required
- number of data movements
- amount of network traffic generated
Algorithm Analysis

- We want to express these measurements as a function of input data size.

- For example, sorting an array of size 10, 10,000, or 10,000,000 will require different amounts of time. We describe the efficiency of a specific sorting algorithm in terms of the size of the input, the length of the array in this case, which we represent by $n$.

- The “time complexity” of an algorithm is expressed as a function of $n$, where $n$ is the size of the input.

examples: $f(n) = n$
$g(n) = \log n$ (in this course, all log are based 2.)
$h(n) = 8n^3 + n \log n$

here $h(n) = 8n^3 + n \log n$ means it takes $h(n)$ time steps to run the algorithm for an input of size $n$. 
Expressing Time Complexity

Given some algorithm, we want a function that counts the number of elementary steps taken by the algorithm.

Obviously, we cannot always predict the input to the algorithm.

In a sequential search for key on array A:

<table>
<thead>
<tr>
<th>input</th>
<th># of times A[ ] is accessed</th>
</tr>
</thead>
<tbody>
<tr>
<td>A[0]=key</td>
<td>1</td>
</tr>
<tr>
<td>A[n/2]=key</td>
<td>n/2</td>
</tr>
<tr>
<td>A[n−1]=key</td>
<td>n</td>
</tr>
</tbody>
</table>
Analysis Measures

- We can examine different measures of the runtime:
  - best-case time
  - average-case time
  - worst-case time

- Most often we examine worst-case time to get an upper bound on the behaviour of the algorithm.

- Thus, we take an algorithm, examine its worst-case (slowest) behaviour, and count the number of steps required.

- We use a model that counts the number of statements that access or modify data.
Guideline for Time Complexity

In order to analyze the time complexity of an algorithm:

- Consider the **worst-case** scenario
- Count the number of time steps (statements that access/modify data) in that scenario
- Express the number as a function of input size \( n \)
Sequential Search

- What is the worst-case time complexity of sequential search?

```java
int n = A.length;  // A 2 steps
int i = 0;
while (i < n) {
    if (A[i] == key)  // B 2 steps (worst case)
        break;
    i++;
}
return i;
```

- Either branch of the if statement requires two steps.
- The while loop iterates \( n \) times (in the worst case).

\[
f(n) = \underbrace{2}_{A} + \underbrace{\sum_{i=0}^{n-1} 2}_{B} = \underbrace{2 + \sum_{i=0}^{n-1} 2}_{C} = 2n + 2
\]

- In the worst case, given an input of size \( n \), our sequential search algorithm requires \( 2n + 2 \) steps.
Selection Sort

- A ‘bad’ but easy way to sort elements of an array

```java
// preconditions: A[0..n-1] is an array of ints

void sort( int [] A ) {
     int n = A.length; 
     for ( int x = 0 ; x < n-1 ; x++ ) {
         for ( int y = x+1 ; y < n ; y++ ) {
                 int temp = A[x];
                 A[x] = A[y];
                 A[y] = temp;
             }
         }
     }
}
```

- What is the worst case?
  - Assume we perform a swap on every iteration.
Selection Sort Analysis

- Again, we count the time steps required in the worst case.

\[
g(n) = \underbrace{1}_A + \sum_{x=0}^{n-2} \left[ \sum_{y=x+1}^{n-1} \underbrace{4}_B \right]_C^D
\]

\[
= 1 + \sum_{x=0}^{n-2} 4(n - x - 1)
\]

\[
= 1 + 4n \sum_{x=0}^{n-2} 1 - 4 \sum_{x=0}^{n-2} x - 4 \sum_{x=0}^{n-2} 1
\]

\[
= 1 + 4n(n - 1) - \frac{4(n - 1)(n - 2)}{2} - 4(n - 1)
\]

\[
= 2(n^2 - n) + 1
\]
Comparing the Running Time of Algorithms

- Different algorithms that solve the same problem have different running times.
- Let's assume a CPU performs one million instructions per second. When the input size is $n = 1000$, the running time of algorithms with the following time complexity varies greatly:

<table>
<thead>
<tr>
<th>time complexity</th>
<th>running time</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\log n$</td>
<td>6.9 microseconds</td>
</tr>
<tr>
<td>$n$</td>
<td>1 millisecond</td>
</tr>
<tr>
<td>$n \log n$</td>
<td>6.9 milliseconds</td>
</tr>
<tr>
<td>$n^2$</td>
<td>1 second</td>
</tr>
<tr>
<td>$n^3$</td>
<td>16 minutes</td>
</tr>
<tr>
<td>$2^n$</td>
<td>$3.4 \cdot 10^{287}$ years</td>
</tr>
<tr>
<td>$n!$</td>
<td>$1.3 \cdot 10^{2554}$ years</td>
</tr>
</tbody>
</table>
Issues with Measuring the Actual Time

- It is not always easy/feasible to measure the time.
- Various factors make it difficult to compare a program’s execution on different computers or with different input sizes.
  - Hardware
  - Operating system
  - Programming language
  - Compiler (and its optimizations).
Advantages of Big-Oh Notation

- Big-Oh notation provides a simpler way to express a function (such as the running time of an algorithm), enabling easier comparison against other functions.

- For example, expressing the running time of an algorithm as $O(n^2)$ is simpler than say:

$$f(n) = \begin{cases} 
200n^2 + 100n \log n + 50n + 400 & \text{if } n < 10 \\
400n + 5000 & \text{if } n \geq 10 \text{ and } n < 100 \\
50n^2 + 2\sqrt{n} + 1 & \text{if } n \geq 100 
\end{cases}$$

- Saying $f(n) \in O(n^2)$ summarizes how quickly $f(n)$ can grow as $n$ increases.
Advantages of Big-Oh Notation (cntd)

- Big-Oh allows us to compare algorithms easily.

\[
\begin{align*}
\text{Algorithm 1} & \\
\text{int } x & = 0; \\
\text{int } y & = 0; \\
\text{for } (\text{int } i=0 ; i<n ; i++) & \{ \\
\text{for } (\text{int } j=0 ; j<n ; j++) & \{ \\
& x += 2; \\
& y += x*2; \\
& \}
\}
\text{f}(n) = 2 + 2n^2 \\
\Rightarrow f(n) \in O(n^2) \text{ and } f(n) \notin O(n)
\end{align*}
\]

\[
\begin{align*}
\text{Algorithm 2} & \\
\text{int } x & = 0; \\
\text{int } y & = 0; \\
\text{for } (\text{int } i=0 ; i<n ; i++) & \{ \\
& x += 2; \\
\text{for } (\text{int } j=0 ; j<n ; j++) & \{ \\
& y += 2*j;
\}
\}
\text{g}(n) = 2 + 2n \\
\Rightarrow g(n) \in O(n)
\end{align*}
\]

- Algorithm 2 is asymptotically faster than Algorithm 1.
Simplifying Functions

- As $n$ grows asymptotically, the largest-order term in function $f(n)$ contributes nearly all of the growth to $f(n)$.

  $$f(n) = 1000n^2 + 5000n + 2000$$

  is surpassed by

  $$g(n) = \frac{1}{1000} n^3$$

  at $n \approx 10^6$.

In other words, as $n$ grows,

- low-order terms don’t matter ($5000n + 2000 \in f(n)$), and
- constants don’t matter ($1000$ in $f(n)$ and $1/1000$ in $g(n)$).

  What matters is that $n^3$ is of higher order than $n^2$. 
Largest Order Terms

- These highest-order terms define an ordering.

As $n$ grows asymptotically, the largest-order terms assume the following ordering, regardless of low-order terms and constants:

\[
1 < \log n < n^{1/4} < n^{1/3} < \sqrt{n} < n < n \log n < n \log^2 n < n \sqrt{n} \\
< n^2 < n^2 \log n < n^3 < n^4 < 2^n < 3^n < 4^n < n! < n^n
\]

- Consequently, $n \in O(n)$ \hspace{1cm} $n \in O(n \sqrt{n})$ \hspace{1cm} $n \in O(n^3)$

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

- Similarly, $15n + 2 \in O(n)$ \hspace{1cm} $15n + 2 \in O(n^2)$ \hspace{1cm} $15n + 2 \notin O(\log n)$
Time Complexity Functions

Here are some functions:

\[
\begin{align*}
    f(n) &= \frac{1}{100} n^3 + 10 n \log^2 n + 5 \sqrt{n} + 10 \\
    g(n) &= 8 n^2 \log n + 2^{4\sqrt{n}} \\
    h(n) &= \begin{cases} 
        n^{3/2} + 2 \log n & \text{if } n \text{ is odd} \\
        5 n^2 + n & \text{if } n \text{ is even}
    \end{cases}
\end{align*}
\]

Here are the same functions expressed in Big-Oh notation:

\[
\begin{align*}
    f(n) &\in O(n^3) \\
    g(n) &\in O(n^2 \log n) \\
    h(n) &\in O(n^2)
\end{align*}
\]
Function $f(n)$ quickly surpasses functions $g(n)$ and $h(n)$.

The largest-order term ($n^3$) dominates.

The effect of constant factors (e.g., $\frac{1}{100}$) and low-order terms decreases as $n$ grows.
We now have an intuition understanding of Big Oh.

How do we actually define \( f(n) \in O(g(n)) \)?

Requiring that \( f(n) \leq g(n) \) doesn’t work since:

- the inequality might not hold for all values of \( n \), and
- \( f(n) \) or \( g(n) \) might have constants we would prefer to ignore.

What we care about is the asymptotic behaviour of \( f(n) \) and \( g(n) \) as \( n \) grows.

How can we define an inequality similar to \( f(n) \leq g(n) \) that:

- omits constant factors, and
- only considers the highest-order term in each function?
Big O Notations

- Informally $f(n) = O(g(n))$ means $f$ is asymptotically smaller than or equal to $g$.

### Definition

$f(n) \in O(g(n)) \iff \exists M > 0, \exists n_0 > 0 \text{ s.t. } \forall n > n_0, f(n) \leq M \cdot g(n)$

- Ignore low-order terms
- Ignore constants
Recall that function $f(n)$ is “big-Oh” of $g(n)$ iff there is some positive constant $M$ such that for any $n$ greater than some initial $n_0$, $f(n) \leq M \cdot g(n)$. 
Big O Summary

- Big-oh is a tool that allows us to compare functions.
- \( O(g(n)) \) is a set of functions.
  - \( O(n^2) \), \( O(\log n) \), \( O(2^n) \) are each a set of functions.
  - Each Big-oh class represents a class of functions for which the definition holds.

Example: \( f(n) = 2n^2 + 10 \) and \( g(n) = n^2 \).

Question: Is \( f(n) \in O(g(n)) \)?
Constant Factors

- The same code can be written in different ways (depending on the programming language)

**Algorithm 1**

```plaintext
int x = 0;
for (int i=0 ; i<n ; i++)
    x += 2 * (A[i] + B[i++]);

running time: $f(n) = 1 + n$
```

**Algorithm 2**

```plaintext
int i = 0;
while (i < n) {
    int j = A[i];
    j += B[i];
    j *= 2;
    x += j;
    B[i]++;
    i++;
}

running time: $g(n) = 2 + 6n$
```

- Counting each step of a statement may increase a constant but it won't affect the asymptotic (Big Oh) running time.
Next topic is recursion.

What is recursion?
- compute one step and calls the same function to solve the remaining subproblem.

Computing $n!$. Recursive or iterative?