

# Complexity Estimation

## Building the Intuition behind Algorithm Analysis

© S.Y. Chen, York University

### Introduction

Asymptotic algorithm analysis is one of the key concepts associated with the study of algorithms and data structures. However, the practical value of this concept can be difficult to extract from a formal mathematical presentation of the material. The following supplement presents a less mathematical approach to algorithm analysis.

Complexity estimation is based on two practical and intuitive questions: How many times? and What does it cost? By analogy to an exact calculation of the running time, the first question represents the summation sign(s) and the second question represents the contents of a summation. Thus, the second question is more accurately stated as What does it cost each time? or What does it cost on average? Further, the answers to these questions can implicitly build in the simplification rules of asymptotic analysis (i.e. ignore constants, find the largest term among sequential components, and multiply embedded components).

### Traditional Analysis

A formal mathematical analysis of code complexity involves assigning cost constants to each expression and then summing these constants for each time that their expression is executed. As a benchmark example, the complexity for selection sort on an array **A** with **n** elements will be calculated. First, a comprehensive mathematical analysis will be used, and later, a simplified (but formal) estimation will be presented.

#### Selection Sort

```
for (int i = 0; i < n; i++)
{
    int lowIndex = i;

    for (int j = i; j < n; j++)
        if (A[j] < A[lowIndex])
            lowIndex = j;

    swap(A, i, lowIndex);
}
```

The complete cost calculation for the above implementation of selection sort can be described by

$$\sum_{i=1}^n \{c_1 + c_2 + \sum_{j=i}^n (c_3 + c_4) + c_5\}$$

where

$c_1$  is the first loop constant

$c_2$  is the cost of the variable initialization

$c_3$  is the second loop constant

$c_4$  is the cost of the branch statement (and assignment)

$c_5$  is the cost of the swap method

Merging constants, the cost calculation proceeds as follows

$$\begin{aligned} & \sum_{i=1}^n \{c_1 + \sum_{j=i}^n c_2\} \\ & \sum_{i=1}^n c_1 + \sum_{i=1}^n \sum_{j=i}^n c_2 \\ & c_1 n + c_2 \sum_{i=1}^n i \\ & c_1 n + c_2 \frac{n(n+1)}{2} \end{aligned}$$

Ignoring constants and lower order terms, the formal analysis arrives at a final asymptotic complexity of  $\mathbf{O}(n^2)$ .

### Complexity Estimation

Since the value of a quantitative result is increased by the addition of a qualitative insight, complexity estimation is presented as an alternative methodology to perform asymptotic analysis of algorithms.

Qualitatively, a summation sign represents how many times a piece of code is executed, and the contents of the summation equation represent what it costs for each execution. The two pertinent questions of algorithm analysis can thus be posed as How many times? and What does it cost?

Returning to the benchmark example, these two questions will be applied to the analysis of selection sort. The first question of How many times? can be answered from both a code and a problem based perspective. Examining the code, the outer loop processes  $n$  times. With respect to the problem of sorting, there are  $n$  elements to be sorted. Either way, the answer to the first question is  $n$  or  $\mathbf{O}(n)$ .

The second question now poses, “For each of the  $n$  elements to be sorted, what does it cost?” Recognizing that the  $O(1)$  initialization and swap method are low-order sequential components, this question will focus on the second for loop. As a loop, a second iteration of How many times? and What does it cost? could be used. However, since the contents of this for loop are clearly  $O(1)$  in cost, determining the number of times that the second loop executes will directly answer (for each of the  $n$  elements to be sorted) What does it cost each time? or What does it cost on average?

The question of What does it cost each time? can again be answered from a code or a problem based perspective. The for loop executes  $n, n-1, n-2, \dots, 3, 2, 1$  times because any of the  $n$  elements can be the smallest element, but only  $n-1$  elements can be the second smallest (after the smallest element has been set aside), and only  $n-2$  elements can be the third smallest. This answer of What does it cost each time? leads to an answer of  $n/2$  or  $O(n)$  for What does it cost on average? Therefore, the overall cost of Selection Sort is  $O(n) * O(n) = O(n^2)$ .

### Selection Sort

How many times?

$n$  elements to sort  $\rightarrow O(n)$

What does it cost (on average)?

$n/2$  elements to examine  $\rightarrow O(n)$

Overall

$O(n) * O(n) = O(n^2)$

As a further simplification, complexity estimation can also be performed line by line.

### Selection Sort

```

for (int i = 0; i < n; i++)           // O(n)
{
    int lowIndex = i;                // O(1)

    for (int j = i; j < n; j++)      // O(n)
        if (A[j] < A[lowIndex])     // O(1)
            lowIndex = j;           // O(1)

    swap(A, i, lowIndex);           // O(1)
}

```

These single-line complexities can then be merged. Sequential terms which answer What does it cost? are added together first. Since these terms are embedded within a structure, they are multiplied by the cost of the term which answers How many times? The asymptotic complexity of selection sort can then be determined using the usual simplification rules.

$$\begin{aligned}
& \mathbf{O}(n) * \{ \mathbf{O}(1) + \mathbf{O}(n)*[\mathbf{O}(1)+\mathbf{O}(1)] + \mathbf{O}(1) \} \\
& = \mathbf{O}(n) * \{ \mathbf{O}(1) + \mathbf{O}(n)*[\mathbf{O}(1)] + \mathbf{O}(1) \} \\
& = \mathbf{O}(n) * \{ \mathbf{O}(1) + \mathbf{O}(n) + \mathbf{O}(1) \} \\
& = \mathbf{O}(n) * \{ \mathbf{O}(n) \} \\
& = \mathbf{O}(n^2)
\end{aligned}$$

Complexity estimation also works for recursive cases. This is useful since recurrence relations can be even more confusing than summation equations. Factorials are used as an example.

#### Recursive calculation of n!

```

int factorial (int n)
{
    if (n <= 1)
        return 1;

    return n * factorial(n-1);
}

```

The recurrence relations for the recursive calculation of n! are

$$\begin{aligned}
\mathbf{T}(1) &= 1 \\
\mathbf{T}(n) &= \mathbf{T}(n-1) + 1
\end{aligned}$$

Using the standard approach, the complexity of the factorial method is derived as follows

$$\begin{aligned}
\mathbf{T}(n) &= \mathbf{T}(n-1) + 1 \\
&= \{ \mathbf{T}(n-2) + 1 \} + 1 \\
&= \{ [ \mathbf{T}(n-3) + 1 ] + 1 \} + 1 \\
&= \dots \\
&= \{ [ \mathbf{T}(1)+1 ] + \dots + 1 \} + 1 \\
&= 1 + 1 + \dots + 1 + 1 \\
&= n * 1 \\
&= \mathbf{O}(n)
\end{aligned}$$

Conversely, complexity estimation has a much simpler path to the same solution.

#### Recursive calculation of n!

How many times?

n levels of recursion  $\rightarrow \mathbf{O}(n)$

What does it cost?

comparison and return  $\rightarrow \mathbf{O}(1)$

Overall

$$\mathbf{O}(n) * \mathbf{O}(1) = \mathbf{O}(n)$$

## Basic Algorithms

As a quick demonstration and reference, the results of complexity estimation are shown for several basic algorithms.

### Insertion Sort

How many times?

n elements to sort  $\rightarrow O(n)$

What does it cost (on average)?

Best, one element to examine  $\rightarrow O(1)$

Worst, n/2 elements to examine  $\rightarrow O(n)$

Average, n/4 elements to examine  $\rightarrow O(n)$

Overall

Best  $\rightarrow O(n) * O(1) = O(n)$

Worst  $\rightarrow O(n) * O(n) = O(n^2)$

Average  $\rightarrow O(n) * O(n) = O(n^2)$

### Bubble Sort

How many times?

n elements to sort  $\rightarrow O(n)$

What does it cost (on average)?

n/2 elements to examine  $\rightarrow O(n)$

Overall

$O(n) * O(n) = O(n^2)$

### Quicksort

How many times?

Best,  $\log n$  levels of recursion  $\rightarrow O(\log n)$

Worst, n levels of recursion  $\rightarrow O(n)$

Average, closer to  $\log n$  levels of recursion  $\rightarrow O(\log n)$

What does it cost?

examine all elements in each partition  $\rightarrow O(n)$

Overall

Best  $\rightarrow O(\log n) * O(n) = O(n \log n)$

Worst  $\rightarrow O(n) * O(n) = O(n^2)$

Average  $\rightarrow O(\log n) * O(n) = O(n \log n)$

## Mergesort

How many times?

$\log n$  levels of recursion  $\rightarrow O(\log n)$

What does it cost?

merge all elements  $\rightarrow O(n)$

Overall

$O(\log n) * O(n) = O(n \log n)$

## Binary Search

How many times?

Best, 1 level of recursion  $\rightarrow O(1)$

Worst,  $\log n$  levels of recursion  $\rightarrow O(\log n)$

Average, closer to  $\log n$  levels of recursion  $\rightarrow O(\log n)$

What does it cost?

examine the middle element  $\rightarrow O(1)$

Overall

Best  $\rightarrow O(1) * O(1) = O(1)$

Worst  $\rightarrow O(\log n) * O(1) = O(\log n)$

Average  $\rightarrow O(\log n) * O(1) = O(\log n)$

## Tree Traversals

How many times?

$n$  nodes to visit  $\rightarrow O(n)$

What does it cost?

process the node  $\rightarrow O(1)$

Overall

$O(n) * O(1) = O(n)$

## Matrix multiplication

How many times?

$n^2$  elements in final matrix  $\rightarrow O(n^2)$

What does it cost?

multiply two  $n$ -length vectors  $\rightarrow O(n)$

Overall

$O(n^2) * O(n) = O(n^3)$

## **A Practical Example**

An advantage of complexity estimation over the traditional mathematical analysis is that it can be applied directly in practical situations (e.g. to word problems). In particular, the design and use of database queries benefits from an understanding of the complexity of

the underlying procedures, but the problem formulations do not map directly to mathematical analysis techniques.

For example, assume that we wish to determine the GPA for a student who has taken  $n$  courses. If there are  $N$  courses in the university database and there are an average of  $m$  students in each course, what is the approximate complexity of this database query?

Using complexity estimation

A database query

How many times?

$n$  courses  $\rightarrow O(n)$

What does it cost?

find a course in the university database  $\rightarrow O(\log N)$

find the student's grade in the course record  $\rightarrow O(\log m)$

Overall

$O(n) * \{ O(\log N) + O(\log m) \}$

$O(n) * O(\log N)$

// assuming that  $N \gg m$

$O(n \log N)$

**Limitations**

Since the two questions of complexity estimation are implicitly connected with a multiplication, they work best when two distinct terms can be identified. Since these two terms for the Towers of Hanoi problem are not conveniently available, complexity estimation can be awkward for this problem. In general, the questions are less suitable in situations which have exponential or factorial complexities.

Examples:

Visiting every node in a complete binary tree with  $n$  levels

How many times?

How many root nodes  $\rightarrow 1$

What does it cost?

How many times?

How many children per node?  $\rightarrow 2$

What does it cost?

How many times?

How many children per node?  $\rightarrow 2$

What does it cost?

...

Overall

$1 * 2 * 2 * \dots * 2 * 2 = O(2^n)$

### Creating every permutation tour for n cities

How many times?

How many choices for the first city?  $\rightarrow n$

What does it cost?

How many times?

How many choices for the second city?  $\rightarrow n-1$

What does it cost?

How many times?

How many choices for the third city?  $\rightarrow n-2$

What does it cost?

...

Overall

$$n * n-1 * n-2 * \dots * 2 * 1 = \mathbf{O(n!)}$$

### **Summary**

Complexity estimation appears to lead to the same final asymptotic complexity as traditional formal analysis techniques. However, its qualitative and intuitive questions can be easier to apply in many situations. It is therefore presented as a useful supplement to learn in addition to traditional complexity analysis methods.