CS 240: Data Structures and Data Management

Chris Thomson

Winter 2013, University of Waterloo

Notes written from Alejandro López-Ortiz's lectures.

1 Introduction & Code Optimization

← January 8, 2013

1.1 Course Structure

The grading scheme is 50% final, 25% midterm, and 25% assignments. There are five assignments, usually due on Wednesday mornings at 9:30 am. There are several textbooks for the course, all of which are optional but recommended. The textbooks cover roughly 80% of the course content. There are also some course notes published online from previous terms, however the lectures will not necessarily follow those notes strictly.

See the course syllabus for more information.

1.2 CS as the Science of Information

So far in our undergraduate careers, computer science has meant programming. However, programming is only a subset of computer science. Computer science is the **science of information**.

What do we want to do with information? We want to:

- **Process it**. Programs ≡ algorithms.
- **Store it**. We want to encode it. This also leads to information theory. Storing information involves data structures, databases, (file) systems, etc., all of which are searchable in some way.
- **Transmit it**. We want to transmit information over networks. This process involves coding theory.
- **Search it**. First, we have to structure it with data structures and/or SQL databases. Information retrieval is the process of searching for textual information instead of numerical information.
- **Mine it**. This involves artificial intelligence and machine learning.
- **Display it**. Information needs to be displayed using computer graphics (CG) and user interfaces (UI, partially related to psychology).
- **Secure it**. Encryption and cryptography are important. Information should also be stored redundantly to prevent harm from catastrophic events.

1.3 Objectives of the Course

- **Study efficient methods of storing, accessing, and performing operations on large collections of data.**

  "Large" is subjective – your data needs to be large enough to justify the additional mental complexity.

  Typical operations:

  – Insert new item.

- "Deleting" data (flagging data as "deleted", not actually deleting the data).
- Searching for data.
- Sorting the data.

Examples of "large data":

- The web.
- Facebook data.
- DNA data.
- LHC (Large Hadron Collider) measurements (terabytes per day).
- All the music from iTunes, for finding duplicate songs, etc.

- **There is a strong emphasis on mathematical analysis.**

  The performance of algorithms will be analyzed using order notation.

- **The course will involve abstract data types (objects) and data structures.**

  We will see examples that have to do with:

  - Databases.
  - Geographic databases.
  - Text searching.
  - Text compression.
  - Web searching.

1.4 Code Optimization & Measuring Performance

Richard Feynman was in charge of the computer group on the Manhattan Project. He made his code run 10x faster.

The initial method to measure performance was to use a **wall clock**. Initial studies looked like this:

Data size	A	B
3	1	3
5	2	9
10	4	10
20	16	11

However, computers were getting better, so we couldn't use wall clocks anymore. Results were not robust enough due to the qucik progression of performance increases in terms of computing power. Moreover, because of the differences in architecture, the same program may have a different execution time on two different computer models.

Idea: rather than comparing algorithms using seconds, we should compare algorithms using the number of operations required for the algorithm to run.

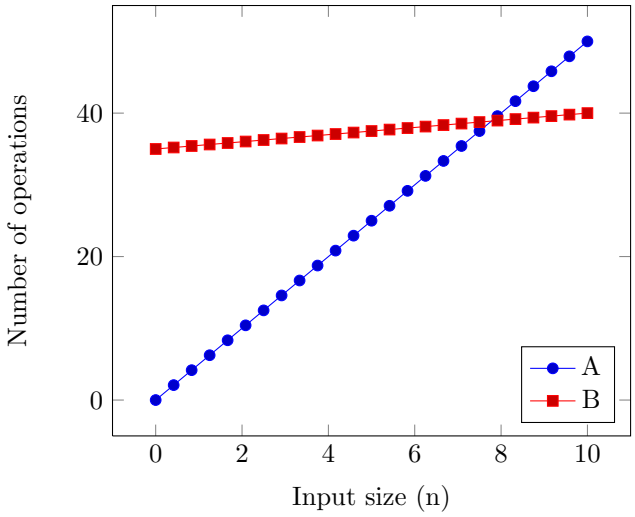

Figure 1: A comparison of two algorithms A and B.

- Express algorithm uisng pseudocode.
- Count the number of primitive operations.
- Plot the number of operations vs. the input size.

Note that A and B are plotted as continuous functions, however they are not actually continuous – we join all of the points for readability purposes only.

In the long run, you may want to use algorithm B even if algorithm A is better in some cases, because the benefits of A will be short lived as n grows.

Hence, comparing the programs has been transformed into comparing functions. We use order notation to measure the long-term growth of functions, allowing us to choose the smaller function, which corresponds to the faster algorithm.

2 Order Notation

← January 10, 2013

The time for algorithm A on input of size n is:

$$\underbrace{2n \log n}_{\text{sort}} + \underbrace{1.5n^3}_{\text{mult}} + \underbrace{22n^2 - 3n}_{\text{additions}} + \underbrace{7}_{\text{setup}}$$

On a different machine, the same algorithm A may be $2n \log n + 9n^3 + 10n^2 - 3n + 7$. These give a **false sense of precision**. These specifics can also be quite costly to determine. In the 1960s, Don Knuth proposed **order notation** as a way to analyze the general quality of algorithms in an accurate, cost- and time-efficient way by ignoring precise runtimes.

2.1 Formal Definitions

Order notation represents algorithms as functions. We say a function $f(n)$ relates to a certain order $g(n)$ using different notation depending on which relation we're interested in.

Relation	Functions
$3 \leq 7$	$f(n) = O(g(n))$
$8 \geq 7$	$f(n) = \Omega(g(n))$
$7 = 7$	$f(n) = \Theta(g(n))$
$3 < 7$	$f(n) = o(g(n))$
$7 > 3$	$f(n) = \omega(g(n))$

Here's an easy way to remember the correspondence between a relation and its order notation symbol: the greedy operators, $\leq$ and $\geq$, are uppercase letters O and Ω. The less greedy operators, $<$ and $>$, are the same letters but in lowercase (o and ω).

Order notation only cares about the long run. An algorithm can violate the relation early – we're interested in its asymptotic behaviour.

Definition 2.1 ($\leq$)**.** $f(n) = O(g(n))$ if there exist constants $c > 0, n_0 > 0$ such that $f(n) \leq c \cdot g(n)$ for all $n \geq n_0$.

Definition 2.2 ($\geq$)**.** $f(n) = \Omega(g(n))$ if there exist constants $c > 0, n_0 > 0$ such that $f(n) \geq c \cdot g(n)$ for all $n \geq n_0$.

Definition 2.3 ($=$)**.** $f(n) = \Theta(g(n))$ if there exists constants $c_1, c_2 > 0$ such that $c_1 \cdot g(n) \leq f(n) \leq c_2 \cdot g(n)$.

The equality relation is effectively sandwiching the algorithm $f(n)$ between two other functions (the squeeze theorem). If it's possible to sandwich the algorithm between two multiples of the same $g(n)$, then $f(x)$ is said to be equal to the order $g(n)$.

Definition 2.4 ($<$)**.** $f(n) = o(g(n))$ if for all $c > 0$ there exists constant $n_0 > 0$ such that $f(n) < c \cdot g(n)$ for all $n \geq n_0$.

Definition 2.5 ($>$)**.** $f(n) = \omega(g(n))$ if for all $c > 0$ there exists constant $n_0 > 0$ such that $f(n) > c \cdot g(n)$ for all $n \geq n_0$.

2.2 Order Notation Examples

Example 2.1. $2n^2 + 3n + 11 = O(n^2)$

We need to show that there exists constants $c > 0$ and $n_0 > 0$ such that $2n^2 + 3n + 11 \leq cn^2$ for all $n \geq n_0$.

Let $c = 3$. Simplifying gets us $3n + 11 \leq n^2$. This holds for $n_0 = 1000$, for instance.

Example 2.2. $2n^2 + 3n + 111 = \Theta(n^2)$

In order to show equality (Θ), we need to show both the $\leq$ and $\geq$ cases. In the previous example, the $\leq$ case was shown. For the $\geq$ case, we need to show $n^2 \leq c \cdot (2n^2 + 3n + 11)$.

Let $c = 1$. Simplifying gets us $n^2 \leq 2n^2 + 3n + 11$, which gives us $0 \leq n^2 + 3n + 11$. This holds for $n_0 = 0$.

Example 2.3. $2010n^2 + 1388 = o(n^3)$

We must show that for all $c > 0$, there exists $n_0 > 0$ such that $2010n^2 + 1388 \leq c \cdot n^3$ for all $n \geq n_0$.

$$\frac{2010n^2 + 1388}{n^3} \overset{?}{\leq} \frac{c \cdot n^3}{n^3} \implies \frac{2010}{n} + \frac{1388}{n^3} \overset{?}{\leq} c$$

There is a **trick to prove f(n) = o(g(n))**: show that $\lim_{n\to\infty} \frac{f(n)}{g(n)} = 0$.

2.3 A note about the use of =

$$\underbrace{2n^2}_{\text{specific function}} \quad \in \quad \underbrace{O(n^2))}_{\text{set of functions}}$$

The use of the equality operator (=) in order notation is not the same as in most other areas of mathematics, since a single element of a set cannot strictly be equal to the set itself (even $a \neq \{a\}$). The = in order notation is not a true equality – it's just notation. The $\in$ operator is semantically more correct.

2.4 Performance of Algorithms

We have made some progression in how we determine the performance of algorithms.

- Comparing single runs.
- Comparing measured curves (multiple runs).
- Produce analytical expressions for the curves.
- Simplify using order notation.

Let's say we have the problem of integrating a mathematical function. Our program numerically integrates the function, and we're trying to analyze the algorithm behind that process. The timing would vary depending on the mathematical function and precise required.

If we conduct multiple runs of our algorithm, we'll get different times. Plotting all of the results (time vs. input size) will result in data that is not a function, because the data fails the vertical line test, since for a given input size we have multiple times.

We need to decide on a convention for these cases where we have multiple time values. In this course, we'll usually look at the worst case, however in some situations examining the best or average cases could be useful.

Finding the average case is hard. To produce an appropriate average, we need an input distribution.

3 Formalism

← January 15, 2013

Definition 3.1. A **problem** is a collection of questions and (correct) answer pairs.

Example 3.1. Informally, the problem is "multiplying two numbers." More formally, the problem could be represented as: (2 x 3, 6), (3 x 4, 12), (1 x 5, 5), etc.

Definition 3.2. An **instance** of a problem is a specific question and answer pair, (Q, A).

Definition 3.3. The **input** of a problem is the question. Example: 3 x 4.

Definition 3.4. The **output** is the only correct answer. Note: there is only one correct answer under this definition. "Multiple" correct answers can always be reduced to some canonical form that represents the one correct answer.

Definition 3.5. An **algorithm** is a mechanical method to produce the answer to a given question of a problem.

Definition 3.6. The **size of the input** is the number of bits, characters, or elements in the question. This definition will vary depending on what's most appropriate for the given question.

Long division in elementary school was the first time a problem's complexity was directly related to the input size. That's not always the case, however.

Definition 3.7. We say an algorithm **solves** a probem if for every question Q it produces the correct answer, A.

Definition 3.8. A **program** is an implementation of an algorithm using a specified computer language.

Example 3.2. We want to sort n numbers. One instance of this problem Q is:

$$(\underbrace{(5,1,3,2,4)}_{\text{Q}},\underbrace{(1,2,3,4,5)}_{\text{A}})$$

For a problem in this form, we'll say the size of Q is $|I| = 5$. Why? Counting the number of elements is the most logical definition in this case.

This course emphasizes algorithms rather than programs. We're computer scientists, so we care about the algorithm and the speed/efficiency of that algorithm.

A problem Π can have several correct algorithms that solve it. Our goal is to find efficient solutions to Π. How do we do that?

1. **Algorithm design**. Find a solution(s) to Π.
2. **Algorithm analysis**. Assess the correctness and efficency of the algorithm.

In this course, we're mostly interested in algorithm analysis. Algorithm design will be covered in more detail in CS 341.

3.1 Timing Functions

A timing function is a function $T_{\mathcal{A}}$ such that:

$$T_{\mathcal{A}} : \{Q\} \to \mathbb{R}^+$$

where $\mathbb{R}^+$ is the set of positive real numbers.

$T_{\mathcal{A}}(Q)$ is the time taken by algorithm $\mathcal{A}$ to compute the answer to Q.

We also want to define a general timing function for a particular problem, regardless of algorithm:

$$\begin{aligned} T(n) &= max\{T_{\mathcal{A}}(Q)\} \\ T_{avg}(n) &= avg\{T_{\mathcal{A}}(Q)\} \\ T_{min}(n) &= min\{T_{\mathcal{A}}(Q)\} \\ &\vdots \end{aligned}$$

If we had two solutions (algorithms) $T_A(n)$ and $T_B(n)$, we could use order notation to compare the quality of A and B.

Note that some problem instances are easier than others to solve, even with the same input size. Most people find it easier to multiply 0×7 than 8×7, for instance, despite those two problems having the same input size.

4 Analysis of Algorithms

In order to analyze an algorithm, we count the number of basic operations that the algorithm performs.

Example 4.1. Let's say our algorithm is to compute $x^2 + 4x$ and assign it to a variable called `result`. That involves seven basic operations:

1. Read x from memory.
2. Compute $x \cdot x$.
3. Assign the result of $x \cdot x$ to a variable, `pr1`.
4. Compute $4 \cdot x$.
5. Assign the result of $4 \cdot x$ to a variable, `pr2`.
6. Compute `pr1` + `pr2`.
7. Assign the result of `pr1` + `pr2` to a variable, `result`.

The number of operations remains the same across machines, but the actual running time of an algorithm will differ from machine to machine (which one of the reasons why we use order notation).

We only count the number of basic operations. There are various definitions of what a "basic" operation actually is (especially with regards to variable assignments), but these are some common operations that are considered to be basic:

- Add numbers of reasonable size (numbers that can be added primitively).
- Multiply numbers of reasonable size (numbers that can multiplied primitively).
- Access an index in an array.
- Store a value in memory (variable assignments).
- Read a value from memory.

Be careful. Some languages disguise complexity well. Just because the syntax is simple doesn't necessarily mean it's a basic operation, and doesn't guarantee that the operation runs in constant time.

4.1 Techniques for Algorithm Analysis

- Straight line programs (no loops). Simply tally up the basic operation count.
- Loops (`for`/`while`). You need to add the number of operations for each pass on the body of the loop.

Example 4.2. Analyze the following algorithm.

for $i = a$ *to* b **do**
| $<$ straight line program $> T_L$
end

This program will run with $\sum_{i=a}^{b} T_L(i)$ operations.

Whenever you have a loop in your algorithm, you should expect to get a summation in your timing function.

Example 4.3. Analyze the following algorithm.

for $x = 0$ *to* 10 **do**
| pr1 = x * x;
| pr2 = 4 * x;
| result = pr1 + pr2;
| print result;
end

This program will run with $\sum_{i=0}^{10} 4 = 44$ operations (depending on your definition of "basic" operations).

Operations like addition and multiplication are primitive for numbers of a reasonable size. Once numbers get large enough to exceed certain limits, we have to resort to some trickery to perform those operations, which adds additional complexity, making them no longer "basic" operations.

We can be a bit sloppy and use the worst case, then say the actual algorithm is $\leq$ what we calculated.

Example 4.4. Analyze the following algorithm. Test 1 (n).

```
sum = 0;
for i = 1 to n do
|  sum = sum + i
end
return sum
```

This program will run with $1 + (\sum_{i=1}^{n} 1) + 1 = 2 + n = \Theta(n)$ operations.

Example 4.5. Analyze the following algorithm.

```
sum = 0;
for i = 1 to n do
|  sum = sum + (i − j)^2;
|  sum = sum^2;
end
return sum
```

This program will run with time:

$$\begin{aligned}
&1 + \sum_{i=1}^{n} \left[\sum_{j=1}^{i} 4 \right] + 1 \\
&= 2 + \sum_{i=1}^{n} 4i \\
&= 2 + 4 \sum_{i=1}^{n} i \\
&= 2 + 4 \cdot \frac{n(n+1)}{2} \text{ (Gauss)} \\
&= 2 + 2n^2 + 2n \\
&= \Theta(n^2)
\end{aligned}$$

We can be pessimistic, assume worst-case scenario, and say that it runs with time:

$$\begin{aligned}
&2 + \sum_{i=1}^{n} n \\
&= O(n^2)
\end{aligned}$$

Note that $O(n^2)$ is a less precise result than $\Theta(n^2)$, but in some cases that's good enough.

Order notation helps make algorithm analysis easier by allowing us to throw away any specific constants, as long as the order remains untouched. The entire analysis process happens within the context of order notation, so you can just start dropping constants immediately.

Keep in mind, it is possible to create a bad over-estimation. You have to be smart about it.

Example 4.6. Analyze the following algorithm. Test 2 (A, n).

```
max = 0;
for i = 1 to n do
    for j = 1 to n do
        sum = 0;
        for k = i to j do
            sum = A[k] + sum;
            if sum > max then
                max = sum;
            end
        end
    end
end
return sum
```

This is the **maximum subsequence problem**. The input is an array of integers $A[1 \ldots n]$, and the output is the consecutive run with the largest sum.

Sample sequence:

2	-4	1	3	-2	8	-1

The running time of this program is $max\left\{ \sum_{i=1}^{j} A[k] \middle| 1 \leq i \leq j \leq n \right\}$.