Lecture 1: Computational Complexity

David Lester

2017

Outline

1. Learning Outcomes
2. Defining the `while` programming language
3. Average-Case and Worst-Case Analysis
4. Order of Growth
5. Properties of “Big-Oh”
6. Lower Bounds

Learning Outcomes

At the end of this week's material you will:

- Be familiar with the simple programming language.

Learning Outcomes

At the end of this week's material you will:

- Be familiar with the simple programming language.
- Be familiar with the methods used to analyze the efficiency of algorithms written in an imperative programming language;

Learning Outcomes

At the end of this week's material you will:

- Be familiar with the simple programming language.
- Be familiar with the methods used to analyze the efficiency of algorithms written in an imperative programming language;
- Understand the 'Big-Oh' notation (and its relatives Ω and Θ);

Learning Outcomes

At the end of this week's material you will:

- Be familiar with the simple programming language.
- Be familiar with the methods used to analyze the efficiency of algorithms written in an imperative programming language;
- Understand the 'Big-Oh' notation (and its relatives Ω and Θ);
- Understand the relationship between these ideas.

Defining `while`

A grammar for `while`:

$$\begin{array}{lcl} S & ::= & x := a \mid \texttt{skip} \mid S_1;\ S_2 \mid \texttt{if}\ b\ \texttt{then}\ S_1\ \texttt{else}\ S_2 \mid \texttt{while}\ b\ \texttt{do}\ S \\ b & ::= & \texttt{true} \mid \texttt{false} \mid a_1 = a_2 \mid a_1 \leq a_2 \mid \neg b \mid b_1 \wedge b_2 \\ a & ::= & x \mid n \mid a_1 + a_2 \mid a_1 - a_2 \mid a_1 \times a_2 \end{array}$$

- Non-terminals: S, b and a. These represent Statements (`Stm`), Boolean Expressions (`BExp`), and Arithmetic Expressions (`AExp`) respectively.

- Non-terminals: S, b and a. These represent Statements (`Stm`), Boolean Expressions (`BExp`), and Arithmetic Expressions (`AExp`) respectively.
- Terminals: keywords (`while` *etc.*), symbols ($+$, *etc.*) and variables x and numerals n.

- Non-terminals: S, b and a. These represent Statements (`Stm`), Boolean Expressions (`BExp`), and Arithmetic Expressions (`AExp`) respectively.
- Terminals: keywords (`while` *etc.*), symbols ($+$, *etc.*) and variables x and numerals n.
- Any of these symbols can be dashed (x') or subscripted (S_1). The numerals n can be thought of as strings of digits.

Extensions

- We will feel free to extend this very simple langauge with extensions such as array expressions and assignments.

Extensions

- We will feel free to extend this very simple langauge with extensions such as array expressions and assignments.
- We will also feel free to use `for-next` to rerpresent some while-loops.

Matrix Addition: a reminder

We can write a matrix addition program to add two $n \times n$ matrices A and B and place the result in C as follows.

```
for $i := 1$ to $n$ do
    for $j := 1$ to $n$ do
        $C[i,j] = A[i,j] + B[i,j]$
```

Analysis

- We count the number of additions for input matrices of size $n \times n$.

Analysis

- We count the number of additions for input matrices of size $n \times n$.
- We begin with the inner loop (which manipulates variable j): this executes n times, and thus performs n additions.

Analysis

- We count the number of additions for input matrices of size $n \times n$.
- We begin with the inner loop (which manipulates variable j): this executes n times, and thus performs n additions.
- Next we observe that the outer loop (which manipulates variable i) also executes n times. This loop incorporates the one for j and thus performs $n \times n$ additions in total.

Average-Case and Worst-Case Analysis

- Suppose that we need to find the index of the first element in a non-empty one-dimensional array bigger than m. The following program performs the task.

```
$index := 0; i := 1;$ while $i \leq n \wedge A[i] \leq m$ do
    $i := i + 1; index := i$
```

Average-Case and Worst-Case Analysis

- Suppose that we need to find the index of the first element in a non-empty one-dimensional array bigger than m. The following program performs the task.

```
$index := 0; i := 1;$ while $i \leq n \wedge A[i] \leq m$ do
    $i := i + 1; index := i$
```

- If we are unlucky, the required element is $A[n]$ and we must loop through all of the elements of A.
  This is the *worst case*.

Average-Case and Worst-Case Analysis

- Suppose that we need to find the index of the first element in a non-empty one-dimensional array bigger than m. The following program performs the task.

```
$index := 0; i := 1;$ while $i \leq n \wedge A[i] \leq m$ do
    $i := i + 1; index := i$
```

- If we are unlucky, the required element is $A[n]$ and we must loop through all of the elements of A.
  This is the *worst case*.
- Alternatively, maybe the *first* element is acceptable. Then we execute the loop test just once.
  This is the *best case*.

Average-Case and Worst-Case Analysis

- Suppose that we need to find the index of the first element in a non-empty one-dimensional array bigger than m. The following program performs the task.

```
$index := 0; i := 1;$ while $i \le n \wedge A[i] \le m$ do
  $i := i + 1; index := i$
```

- If we are unlucky, the required element is $A[n]$ and we must loop through all of the elements of A.
  This is the *worst case*.
- Alternatively, maybe the *first* element is acceptable. Then we execute the loop test just once.
  This is the *best case*.
- Depending on the precise distribution of the elements of the matrix A we may be able to give an *average case*.

Average-Case and Worst-Case Analysis

- Suppose that we need to find the index of the first element in a non-empty one-dimensional array bigger than m. The following program performs the task.

```
$index := 0; i := 1;$ while $i \leq n \wedge A[i] \leq m$ do
    $i := i + 1; index := i$
```

- If we are unlucky, the required element is $A[n]$ and we must loop through all of the elements of A.
  This is the *worst case*.
- Alternatively, maybe the *first* element is acceptable. Then we execute the loop test just once.
  This is the *best case*.
- Depending on the precise distribution of the elements of the matrix A we may be able to give an *average case*.
- For example if the elements of A are independently normally distributed with mean m and a non-zero variance, and are sorted into ascending order, we would expect to have to search just half the

Important

- Be very careful about assuming lists are random if they are ordered. performance.

Important

- Be very careful about assuming lists are random if they are ordered. performance.
- **Very Important** Be very careful about assuming your data is random. For example, it is very easy to repeatedly sort lists which are already partly or fully sorted.

Order of Growth

- On different machines, we will get a different performance for a particular algorithm, but unless it has a very strange instruction set – say a one cycle quicksort instruction – there will be a relationship between the size of the input and the length of time taken to run the algorithm.

Observations

Observation

This observation allows us to ignore the constant scaling factor for the time taken.

Example

Example

Consider the two functions, each of type $\mathbb{N} \to \mathbb{N}$:

$$\begin{aligned} f(n) &= n \\ g(n) &= n^2 \end{aligned}$$

It does not matter which value of n we select, it will always be the case that $n \leq n^2$. We say that n^2 *dominates* n.

Example

Example

Suppose that the two functions (again $\mathbb{N} \to \mathbb{N}$) this time are:

$$\begin{aligned} f(n) &= 100n \\ g(n) &= n^2 \end{aligned}$$

Now, if $n < 100$ then $f(n) > g(n)$. For example at $n = 10$ we get $f(n) = 1000$ and $g(n) = 100$. But *eventually* $g(n)$ will be bigger than $f(n)$. In this case "eventually" means for any value of $n > 100$; for other functions this may differ.

Eventually Dominating Functions

Definition

We say the function g *eventually dominates* function f, whenever there exists $k : \mathbb{N}$ such that:

$$\forall\ (n : \mathbb{N}).\ n > k \Rightarrow g(n) > f(n)$$

In other words, whenever n is greater than k, and we have $g(n) > f(n)$, then we can say that g eventually dominates f.

Observation

This observation allows us to ignore functions that are eventually dominated by another function.

Puting Observations 4.1 and 4.5 together we have arrived at the following useful conclusion for polynomial complexities: if $T(n) = a_k n^k + a_{k-1} n^{k-1} + \cdots + a_2 n^2 + a_1 n + a_0$ then it will eventually be dominated by Cn^k, for some $C > 0$. If, in addition, we do not care about the scaling constant we can simple talk about the function n^k.

Definition

If $f, g : \mathbb{N} \to \mathbb{R}_{\geq 0}$ and $f \in O(g)$, then there exists $k \in \mathbb{N}$ and $C > 0$ such that for all $n > k$:

$$f(n) \leq Cg(n)$$

Properties of “Big-Oh”

In this section we will demonstrate some of the properties of the notation. You should be aware that many people are informal and sloppy in their use of the notation, so be prepared for this. In particular, instead of writing the function properly you will see things such as $O(1)$ and $O(n)$. These are actually shorthand for the polynomial functions $f(n) = 1$ and $f(n) = n$ respectively.

Reflexive Property

Lemma

$f \in O(f)$

Proof

Pick $k = 0$ and $C = 1$. Then for all $n \in \mathbb{N}$:

$$f(n) = Cf(n) \leq Cf(n)$$

$\square$

Let us next show that the constant functions $O(1)$ are a strict subset of the linear functions $O(n)$.

Lemma

$$O(1) \subset O(n)$$

i.e. $O(1)$ is a strict subset of $O(n)$.

The proof is in the notes.

Lemma

Suppose $g(n)$ is a polynomial, i.e.

$$g(n) = a_k n^k + a_{k-1} n^{k-1} + \cdots + a_2 n^2 + a_1 n + a_0$$

Provided that the leading coefficient $a_k > 0$, then the set $O(g)$ is the same set as $O(n^k)$

Lemma

$$O(1) \subset O(\log n) \subset O(n)$$

(In Computer Science we usually take $\log(n)$ to be $\log_2(n)$.)

Lower Bounds

- Related to the“Big-Oh” notation: is the “Big-Omega” notation.

Lower Bounds

- Related to the"Big-Oh" notation: is the "Big-Omega" notation.
- In saying $f \in \Omega(g)$ we are saying that the function f *eventually dominates* g; *i.e.* g is a *lower bound* to f.

Definition of $\Omega(f)$

Definition

If $f, g : \mathbb{N} \to \mathbb{R}_{\geq 0}$ and $f \in \Omega(g)$, then there exists $k \in \mathbb{N}$ and $C > 0$ such that for all $n > k$:

$$Cg(n) \leq f(n)$$

Important

- *Algorithms* have an upper bound on their execution time ("quicksort is average-case $O(n \log n)$"); and

Important

- *Algorithms* have an upper bound on their execution time ("quicksort is average-case $O(n \log n)$"); and
- *Problems* have a lower bound ("sorting *must* be $\Omega(n)$ since we must compare each element at least once").

Algorithmic Gaps

- If the best algorithm for a problem has the same as the problem complexity then we have – in some sense – a near ideal program.

Algorithmic Gaps

- If the best algorithm for a problem has the same as the problem complexity then we have – in some sense – a near ideal program.
- Alternatively – and much more common – there is an *Algorithmic Gap* in that the best algorithm is worse than the problem's lower bound.

Binary Search

- Search is an example where we *know* that there is no algorithmic gap.

Binary Search

- Search is an example where we *know* that there is no algorithmic gap.
- To search a telephone directory, we translate the name to an integer, *e.g.* "Dave" would become $1,147,237,989$.

Binary Search

- Search is an example where we *know* that there is no algorithmic gap.
- To search a telephone directory, we translate the name to an integer, *e.g.* “Dave” would become $1,147,237,989$.
- Using `while` we set variable x to 1147237989 (the name).

Binary Search

- Search is an example where we *know* that there is no algorithmic gap.
- To search a telephone directory, we translate the name to an integer, *e.g.* "Dave" would become $1,147,237,989$.
- Using `while` we set variable x to 1147237989 (the name).
- To return the answer we set variable y to 55726, the phone number.

Binary Search (ctd.)

- Somewhere in the program we must have an assignment:

$$y := 55726$$

Binary Search (ctd.)

- Somewhere in the program we must have an assignment:

$$y := 55726$$

- Before we get to this assignment, we must select a path: this involves the use of `if then else` construct.

Binary Search (ctd.)

- Somewhere in the program we must have an assignment:

$$y := 55726$$

- Before we get to this assignment, we must select a path: this involves the use of `if then else` construct.
- To minimize the number of comparisons, we need to make the first comparison split the space into 2 equal parts.

Binary Search (ctd.)

- Somewhere in the program we must have an assignment:

$$y := 55726$$

- Before we get to this assignment, we must select a path: this involves the use of `if then else` construct.
- To minimize the number of comparisons, we need to make the first comparison split the space into 2 equal parts.
- We repeat this procedure, and after 20 comparisons we can have over $1,000,000$ possible telephone numbers stored.

Binary Search (ctd.)

- Somewhere in the program we must have an assignment:

$$y := 55726$$

- Before we get to this assignment, we must select a path: this involves the use of `if then else` construct.
- To minimize the number of comparisons, we need to make the first comparison split the space into 2 equal parts.
- We repeat this procedure, and after 20 comparisons we can have over $1,000,000$ possible telephone numbers stored.
- For searching a telephone directory, the time-complexity of the best known algorithm is $O(\log n)$, which is the *same as* the lower bound for the problem which is $\Omega(\log n)$.

Conclusion

- We have introduced a very simple imperative programming language.

Conclusion

- We have introduced a very simple imperative programming language.
- We have seen how to analyse simple programs with loops and give the complexity.

Conclusion

- We have introduced a very simple imperative programming language.
- We have seen how to analyse simple programs with loops and give the complexity.
- We have discussed algorithmic complexity, and why it is useful.

Conclusion

- We have introduced a very simple imperative programming language.
- We have seen how to analyse simple programs with loops and give the complexity.
- We have discussed algorithmic complexity, and why it is useful.
- We have introduced lower bounds for problems and discussed *algorithmic gaps*.