Asymptotic Analysis: When the product rule isn’t tight

Russell Impagliazzo and Miles Jones, thanks Janine Tiefenbruck

April 8, 2016
1. When is the product rule not tight?
Suppose a loop will be executed at most $T_1$ times, and each time, the body (the inner loop) gets executed. If we’ve already analyzed the body as taking time $O(T_2)$ in the worst case, we know that the total time for the loop is no more than the product of the number of iterations and the time for the body, i.e., $O(T_1 \times T_2)$.

Remember that $O$ is an upper bound, so it will not always be tight.
When is the product rule tight?

Let’s say in the $k$’th iteration of the loop, the body takes some $t_k$ time. Then an exact formula for the total time for all the runs of the body is:

$$\sum_{k=1}^{k=T_1} t_k.$$  

The product rule uses the fact that each $t_k \leq T_2$, where $T_2$ the worst-case time for any iteration, to say this sum is at most $T_1 \times T_2$, the number of terms times the biggest term.

- This is a good estimate when many of the terms are pretty close to the bound.
- A good rule of thumb is to look at the middle term.
- But when this isn’t true, when most runs are much faster than the worst-case, we can often give a better upper bound on the total time.
Example: Do sorted arrays intersect?

The problem:
Given two sorted arrays $A[1, \ldots, n]$ and $B[1, \ldots, n]$, determine if they intersect, i.e. if some $A[I] = B[J]$.

A solution:
- We use a linear search to see if $B[1]$ is anywhere in $A$.
- In general, since $B[J] > B[J - 1]$, we can start the search for the next $B[J]$ where our search for $B[J - 1]$ left off.
Intersecting sorted arrays

\textit{Intersect}(A[1, \ldots, n], B[1, \ldots, n])

1. \( I \leftarrow 1 \).
2. FOR \( J = 1 \) TO \( n \) DO:
3. \hspace{1em} WHILE \( B[J] > A[I] \) AND \( I \leq n \) DO: \( I++ \)
4. \hspace{1em} IF \( I > n \) THEN Return \textit{False}
5. \hspace{1em} IF \( B[J] = A[I] \) THEN Return \textit{True}
6. Return \textit{False}

In the worst-case, the inside WHILE loop can run \( n \) times, and the outside FOR loop has \( n \) iterations. Thus, the product rule gives an upper bound of \( O(n^2) \) time. But this isn’t tight.
Intersecting sorted arrays

\[
\text{Intersect}(A[1, \ldots, n], B[1, \ldots, n])
\]

1. \( I \leftarrow 1. \)
2. FOR \( J = 1 \) TO \( n \) Do:
3. \hspace{0.5cm} WHILE \( B[J] > A[I] \) AND \( I \leq n \) DO: \( I \) ++
4. \hspace{0.5cm} IF \( I > n \) THEN Return \( False \)
5. \hspace{0.5cm} IF \( B[J] = A[I] \) THEN Return \( True \)
6. \hspace{0.5cm} Return \( False \)

The inside WHILE loop can run \( n \) times ONCE, but then the rest of the time, it won’t be done at all. In fact, except for the last iteration in every FOR loop, every time line 3 is executed, \( I \) is incremented, and if \( I \) reaches \( n + 1 \), the program terminates. So line 3 only is run \( 2n \) times total, which makes the entire time for this algorithm \( O(n) \).
Here’s an example where we change what we are counting as a basic operation. Usually, we count arithmetic operations such as addition or multiplication as constant time. However, there are situations where that’s unreasonable. CPU’s can perform operations on a fixed size integer, usually 64 bits. When numbers are more than 64 bits long, floating point operations estimate the values of numbers, but are not exact. Can you think of an application where we need to perform operations on very large numbers and it has to be exact?
When the numbers are very long, and we need to perform operations exactly, we can think of the binary representations as being given as an *array of bits*. Then operations on each bit become constant time, but addition overall isn’t necessarily constant time.

Say we are adding $x$ which in binary is written $x_{n-1}...x_0$ and $y = y_{n-1}...y_0$, (If one number is smaller, we can add 0’s at the start to make them equal.) Then the value of $x$ is given by the formula: $x = \sum_{i=0}^{n} x_i$

What do you guess the time complexity of the standard grade school addition algorithm is?

A  $O(1)$
B  $O(n)$
C  $O(n \log n (\log \log n)^2)$
D  $O(n^2)$. 
Add\( (x[0...n-1], y[0..n-1]) \)

1. \( c \leftarrow 0 \) \{carry bit\}
2. For \( I = 0 \) TO \( n - 1 \) do:
   3. \( z_I \leftarrow (c + x_I + y_I) \mod 2 \)
   4. \( c \leftarrow c + x_I + y_I \div 2 \)
5. \( z_n \leftarrow c \)
6. Return \( z_n...z_0 \).
Binary addition, cont.

1. \( c \leftarrow 0 \) \{carry bit\}

2. For \( I = 0 \) TO \( n - 1 \) do:

3. \( z_I \leftarrow (c + x_I + y_I) \mod 2 \) \{O(1)\}

4. \( c \leftarrow c + x_I + y_I \div 2 \) \{O(1)\}

5. \( z_n \leftarrow c \)

6. Return \( z_n \ldots z_0 \).

The inside of the For loop is constant time, since these operations are on single bits, not arbitrarily large numbers.
Binary addition, cont.

1. \( c \leftarrow 0 \{ \text{carrybit} \} \{ O(1) \} \)
2. For \( I = 0 \) TO \( n - 1 \) do: \( \{ O(n) \} \)
3. \( z_I \leftarrow (c + x_I + y_I) \mod 2 \{ O(1) \} \)
4. \( c \leftarrow c + x_I + y_I \div 2 \{ O(1) \} \)
5. \( z_n \leftarrow c \{ O(1) \} \)
6. Return \( z_n \ldots z_0 \).

The loop is repeated \( n \) times, so that makes it \( O(n) \). The other lines are constant time, so the whole algorithm is \( O(n) \).
Say we are adding $x$ which in binary is written $x_{n-1}...x_0$ and $y = y_{m-1}...y_0$, and we’ll assume $m \leq n$. We’ll use this as an example of an analysis in terms of two parameters, both $n$ and $m$. What do you guess the time complexity of the standard grade school multiplication algorithm is?

A  $O(1)$  
B  $O(n + m)$  
C  $O((n + m) \log n(\log \log n)^2)$  
D  $O(nm)$.  

Let Shift append a 0 to the end of a binary string, and move every bit over one position.

Multiply($x[0..n-1],y[0..m-1]$)

\[
    z \leftarrow 0;
\]

For $l = 1$ TO $m - 1$ do:

IF $y_l = 1$ THEN Add($z, x$)

Shift($x$)

Return $z$
Multiply($x[0..n-1],y[0..m-1]$)

\[ \begin{align*}
    z & \leftarrow 0; \\
    \text{For } I = 1 \text{ TO } m - 1 \text{ do:} \\
    \quad \text{IF } y_I = 1 \ \text{THEN } \text{Add}(z,x)\{O(n)\} \\
    \quad \text{Shift}(x)\{O(n)\} \\
    \text{Return } z
\end{align*} \]

of $x$ in the $I$’th iteration is $n + I$ bits. After $I = 0$, $z$ might have $n$ bits. If we add two numbers, the result is at most one bit larger than the max length. So $z$ also has at most $n + I \leq n + m$ bits total. Thus, the time in line 3 is $O(n + m) = O(n)$ since $m \leq n$. Shift is also $O(n)$ for the same reason.
Multiply(x[0..n-1],y[0..m-1])

    z ← 0;
    For I = 1 TO m − 1 do:
        IF y_I = 1 THEN Add(z, x){ O(n) }
        Shift(x){ O(n) }
    Return z

Then the $O(n)$ line is repeated $m$ times in the loop, making the total time $O(nm)$. 
The best known algorithm for multiplication of two $n$ bit integers is:

A  $O(1)$
B  $O(n + m)$
C  $O((n) \log n(\log \log n)^2)$
D  $O(n^2)$. 