1 Introduction

We introduce techniques for analyzing the runtime of functional programs. You should already be familiar with the basic concepts, especially “big-O” notation but we give a brief recap of the main ideas. We focus here mainly on “work”, an estimate of the runtime on a sequential processor. Later in the semester we will introduce “span”, an estimate of the runtime assuming parallel evaluation of independent sub-expressions and an unlimited supply of processors. The math concepts and tools (recurrences, and big-O notation) are relevant for work and span.

- We discuss the relationship between work and the number of steps ML takes to evaluate an expression.

- We show how to obtain a recurrence relation for the runtime of an ML function when applied to an argument with a given size.

- We show how to find exact solutions to recurrences, or an asymptotic approximation when an exact solution is not needed or not feasible.

- We list solutions for some common recurrence relations.

- Sometimes the efficiency of a function can be improved by using an “accumulator”, or computing extra information.

- We show how mathematical insight may also lead to more efficient code.

- For good measure, we include in the self-test some correctness proofs as well as questions about efficiency.
2 Asymptotic analysis

We focus on an asymptotic analysis of the work done during evaluation of expressions. This kind of analysis predicts how long it will take to run your code on really big inputs, without actually running it. It is one of the main tools used to choose between different algorithms for the same problem. Underlying this kind of analysis is the assumption that primitive operations (such as arithmetic and boolean operators, or cons-ing an item onto a list) take constant time and that we don’t care about (and don’t need to know) the precise value of these constants. We mainly care about what happens with “large” arguments, because one could easily re-design a function to do something fancy for a few small arguments, but that would likely have no significant effect on how the function works on large arguments. That’s what asymptotic analysis is all about: estimating the runtime for large arguments.

We want our analysis to be robust, so “work” isn’t going to be expressed in units like micro-seconds, seconds, minutes, hours, or days; it’s not sensible to claim that your piece of code takes 33 milliseconds to produce the result 42. Rather we will show how to prove assertions like “the work to evaluate \( f(L) \) is proportional to \( n^2 \), when \( L \) is a list of integer values of length \( n \)”. This will enable us to make predictions about the way code speeds up or slows down when we change one piece for another. Implicitly, work represents the number of “basic” steps needed to evaluate the piece of code, and we usually express it as some function of the “size” of some parameter (typically, the argument to some relevant function). When we say “is proportional to \( n^2 \)” we mean that there is some constant \( c \) such that for large argument size \( n \) the runtime is less than or equal to \( iscn^2 \).

big-O classification

Asymptotic analysis is based on big-O classifications: \( O(1) \) or “constant time”; \( O(n) \) or “linear”; \( O(n^2) \) or “quadratic”; \( O(log \ n) \), or “logarithmic”; and so on. As we have said, big-O abstracts away from constant factors. So an algorithm with running time proportional to \( 50000n^3 \) is \( O(n^3) \) and so is an algorithm with running time \( 2n^3 \). In fact constant factors sometimes do make a difference, practically, especially for low input sizes; but usually the behavior when inputs get very large is more significant. And we would clearly prefer a running time of \( 50000n^3 \) to a running time of \( 2^n \), since \( 2^n > 50000n^3 \) for all large enough values of \( n \). Thus we say that \( O(n^3) \) is better than or
faster than $O(2^n)$.

More rigorously, for two functions $f, g$ of type $\texttt{int} \rightarrow \texttt{int}$ we say that “$f$ is $O(g)$” if there is a (real-valued) constant $c$ and an integer $N$ such that for all $n \geq N$, $|f(n)| \leq c|g(n)|$.

When the values of $f(n)$ and $g(n)$ are always non-negative (e.g. when they represent running times of code fragments!) we can elide the absolute value signs and just say “$f$ is $O(g)$” when “for all $n \geq N$, $f(n) \leq c \cdot g(n)$”. Similarly we only usually care about non-negative values of $n$, because in our analysis $n$ usually stands for some measure of “argument size”.

We often say “for sufficiently large $n$” as an abbreviation for “for all $n \geq N$, for some $N$”.

We usually simplify and write something like $30n^2 + 4000n + 1$ is $O(n^2)$, rather than naming the functions (e.g. “let $f(n) = 30n^2 + 4000n + 1$...”).

We may take advantage of well known results about big-O notation, for instance the fact that “constants don’t matter”. At the end of the notes for today we summarize some key results.

Comments

We say that “$f$ is $O(g)$”. Some people use “$f = O(g)$” or “$f \in O(g)$”. Sometimes we’ll write something like $f(n) = O(n) + n^2$ to mean that there is a function $g(n)$ belonging to $O(n)$ such that $f(n) = g(n) + n^2$. Note that in this case it follows that $f$ is actually $O(n^2)$.

Another way to think of $O(f)$ is as a set of functions — the functions that satisfy the definition of $O(f)$. Thus it is common to see notation like $O(n^2) \subset O(n^3)$ to indicate the (true) fact that every function that is $O(n^2)$ is also $O(n^3)$. The use of the “proper inclusion” symbol $\subset$ emphasizes the (also true) fact that it isn’t true that every function that is $O(n^3)$ is also $O(n^2)$. Indeed, it is pretty obvious that the function $f(n) = n^3$ is $O(n^3)$ but not $O(n^2)$. We may also write (true) statements such as $O(n^2) = O(n^2 + 543n + 42)$ and $O(n^2 + 2^n) = O(2^n)$ to indicate when two big-O classes contain exactly the same functions.
3 Examples

In these examples we sometimes omit the type annotations that we have insisted on previously, because we want to focus on runtime analysis and all of our code is known to be well-typed; you should continue to obey the requirements in your own code development, until we tell you otherwise! As an exercise, you can figure out how to put type annotations into our examples. In any case, our code runs perfectly well without the extra type information. As we will see shortly, ML can do a lot of type inference in the background.

We also relax the requirement to include REQUIRES and ENSURES comments, again because the focus here is on runtime, not on correctness. Nevertheless we try to give clear informal specifications, and we indicate how you could prove correctness. Moreover, sometimes we need to know something about the applicative behavior of a function to justify our runtime analysis; in such cases it will be vital to have a good specification for that function! Indeed, in many of the examples, we do make assertions about the applicative behavior of our functions, which talk about the results produced by a function when applied to an argument. You can use the proof techniques from the previous lectures to fill in the details, if you so desire. (And some of the examples will already have been covered in class.)

3.1 Powers of 2

Here is a simple ML function \( \text{exp} : \text{int} \rightarrow \text{int} \) for calculating powers of 2:

\[
\text{fun exp (n:int):int = if n=0 then 1 else 2 * exp (n-1)}
\]

It is easy to prove by induction that for all \( n \geq 0 \), \( \text{exp} \ n = 2^n \).

Let \( W_{\text{exp}}(n) \) be the running time (or “work”) of \( \text{exp} \ n \), for \( n \geq 0 \). We assume (as usual) that arithmetic and boolean operations take constant time. It should then be clear from the structure of the function definition that there are (non-negative) constants \( c_0, c_1 \) such that

\[
\begin{align*}
W_{\text{exp}}(0) &= c_0 \\
W_{\text{exp}}(n) &= c_1 + W_{\text{exp}}(n - 1), \text{ for } n > 0.
\end{align*}
\]

Think of \( c_0 \) as representing the number of basic operations needed to evaluate \( \text{exp}(0) \) down the value 1, and \( c_1 \) as the number of basic operations needed when \( n > 0 \), to get from \( \text{exp}(n) \) to \( 2 \times \text{exp}(n - 1) \). (Using the evaluation
rules for our programming language we could actually calculate what these constants are: the number of $\Rightarrow$ steps taken, in each case. But the details don’t really matter so it is convenient just to use named constants with unspecified values in the analysis.)

Using this recurrence relation it is easy to prove, by induction on $n$, that for all $n \geq 0$, $W_{\text{exp}}(n) = n \cdot c_1 + c_0$. Exercise: prove this.

This result, that for all $n \geq 0$, $W_{\text{exp}}(n) = n \cdot c_1 + c_0$, is called a closed form solution of the recurrence relation. This closed form makes it obvious that $W_{\text{exp}}(n)$ is linear in $n$.

It is also easy to show, using this closed form, that $W_{\text{exp}}(n)$ is $O(n)$. Here is a sketch of the details. We know that there are (positive) constants $c_0$ and $c_1$ such that $W_{\text{exp}}(n) = n \cdot c_1 + c_0$, for all $n \geq 0$. Pick $c$ to be $c_1 + 1$ and let $N = c_0$. Then for all $n \geq N$ we have

$$W_{\text{exp}}(n) = n \cdot c_1 + c_0 \leq n \cdot (c_1 + 1) = c \cdot n.$$ 

Thus, according to the definition of big-O, $W_{\text{exp}}(n)$ is $O(n)$. In other words, the running time for $\text{exp } n$ is linear in $n$.

Actually, it can be convenient to make a simplifying assumption about these “unknown” constants. It should be clear that for any non-negative constants $c_0$ and $c_1$, the function $f(n) = n \cdot c_1 + c_0$ is $O(n)$. The choice of constants makes no difference to this fact. So we could have made an arbitrary decision to choose $c_0 = c_1 = 1$ and taken the recurrence defining $W_{\text{exp}}$ to be

$$W_{\text{exp}}(0) = 1 \quad W_{\text{exp}}(n) = 1 + W_{\text{exp}}(n - 1), \text{ for } n > 0.$$ 

We would have then been able to show that $W_{\text{exp}}(n) = n + 1$ for $n \geq 0$, and hence that $W_{\text{exp}}(n)$ is $O(n)$ as before.

Having shown that $W_{\text{exp}}(n) = n \cdot c_1 + c_0$, now let’s see how that connects with evaluation steps and how we can do more sophisticated runtime analysis using this information.

Firstly, assuming that the constants were chosen to match up with how many $\Rightarrow$ evaluation steps really happen, we can safely say that:

For all values $n \geq 0$,

$$\text{exp}(n) \Rightarrow^{(nc_1+c_0)} k, \quad \text{where } k \text{ is the value of } 2^n$$
Here we annotate the evaluation symbol not with $\ast$ but with an indication of the number of steps taken, i.e. we write $\Rightarrow (m)$ where $m$ is the number of steps, rather than $\Rightarrow \ast$, which we used to mean in some finite number of steps. (Also I use the nicer looking notation $\Rightarrow$ instead of $\Rightarrow$ simply because it is available in LaTeX math mode!)

Using the above property, we can now answer questions about code that uses the function $\texttt{exp}$. For example, when $n \geq 0$, what is the work needed to evaluate the expression $\texttt{exp(exp(n))}$? We can figure this out as follows, using the evaluation rules to guide us. We have:

\[
\begin{align*}
\texttt{exp(exp(n))} & \Rightarrow^{(1)} (\texttt{fn } \ldots)(\texttt{exp(n)}) \\
& \Rightarrow^{(nc_1+c_0)} (\texttt{fn } \ldots)(k) \\
& \Rightarrow^{(kc_1+c_0-1)} m
\end{align*}
\]

where $k$ is the integer value of $2^n$ and $m$ is the integer value of $2^k$, so $m = 2^{2^n}$. The third line in this derivation is justified because we know from the above property that

\[
\texttt{exp(k)} \Rightarrow (kc_1+c_0) m,
\]

and this computation begins with the step

\[
\texttt{exp(k)} \Rightarrow^{(1)} (\texttt{fn } \ldots)(k),
\]

so the rest of this computation looks like

\[
(\texttt{fn } \ldots)(k) \Rightarrow^{(kc_1+c_0-1)} m.
\]

So the overall conclusion here is that the number of steps to evaluate $\texttt{exp(exp(n))}$, the work for this expression, is

\[
1 + (nc_1+c_0) + (kc_1+c_0-1) = nc_1 + 2^n c_1 + c_0.
\]

In this formula the linear term grows less quickly than the exponential term, and we therefore deduce that the work for this expression is $O(2^n)$.

### 3.2 Powers of 2, faster

Now let’s define a (more efficient) function $\texttt{fastexp : int -> int}$ that takes advantage of some simple mathematical facts about powers of 2. Specifically whenever $n > 0$, either $n$ is even, and $2^n = (2^{n \div 2})^2$; or $n$ is odd, and $2^n = 2 \ast 2^{n-1}$.
fun square (x:int):int = x*x;

fun fastexp (n:int):int = 
  if n = 0 then 1 else 
    if (n mod 2 = 0) then square (fastexp (n div 2)) 
    else 2 * fastexp (n-1)

Again it is easy to prove that for all $n \geq 0$, \texttt{fastexp} $n = 2^n$.

Now let $W_{\text{fastexp}}(n)$ be the runtime of \texttt{fastexp} $n$, for $n \geq 0$. Again the structure of the function definition tells us that there are constants $k_0, k_1, k_2$ such that:

\[
\begin{align*}
W_{\text{fastexp}}(0) &= k_0 \\
W_{\text{fastexp}}(n) &= k_1 + W_{\text{fastexp}}(n \div 2) & \text{if } n > 0 \text{ and } n \text{ even} \\
W_{\text{fastexp}}(n) &= k_2 + W_{\text{fastexp}}(n - 1) & \text{if } n > 0 \text{ and } n \text{ odd}
\end{align*}
\]

Hence, because $n - 1$ is even and non-negative when $n$ is odd and positive, and in such a case $(n - 1) \div 2$ is equal to $n \div 2$, we actually have:

\[
\begin{align*}
W_{\text{fastexp}}(0) &= k_0 \\
W_{\text{fastexp}}(n) &= k_1 + W_{\text{fastexp}}(n \div 2) & \text{if } n > 0 \text{ and } n \text{ even} \\
W_{\text{fastexp}}(n) &= k_2 + k_1 + W_{\text{fastexp}}(n \div 2) & \text{if } n > 0 \text{ and } n \text{ odd}
\end{align*}
\]

Since we only care about the asymptotic runtime, we lose no generality by expanding out the case for $n = 1$, setting all constants to 1, and working with the recurrence relation given by

\[
\begin{align*}
T_{\text{fastexp}}(0) &= 1 \\
T_{\text{fastexp}}(1) &= 1 \\
T_{\text{fastexp}}(n) &= 1 + T_{\text{fastexp}}(n \div 2) & \text{for } n > 1.
\end{align*}
\]

$T_{\text{fastexp}}$ defined this way is obviously not the same function as $W_{\text{fastexp}}$ as given above, but it can be shown that these two functions have the same asymptotic behavior. It’s much easier to find a closed form for $T_{\text{fastexp}}$.

Indeed this recurrence for $T_{\text{fastexp}}$ is \textit{exactly the same recursive pattern} as we used in lab to define the logarithm function $\log : \texttt{int} \rightarrow \texttt{int}$, and we already proved in lab that this function computes logarithms in base 2. So we can get a closed form for $T_{\text{fastexp}}(n)$ immediately: For all $n \geq 1$, $T_{\text{fastexp}}(n) = \log_2(n)$. Recall that $\log_2 n$ is the largest non-negative integer $k$ such that $2^k \leq n$.  

This doesn’t imply that $W_{fastexp}(n)$ is also equal to $\log_2(n)$ — it couldn’t be, because its recurrence relation mentions $k_0, k_1, k_2$. But we said that $W_{fastexp}$ and $T_{fastexp}$ have the same asymptotic behavior. That means that $W_{fastexp}(n)$ is in the same $O$-class as $T_{fastexp}(n)$. Hence $W_{fastexp}(n)$ is $O(log_2 n)$.

Recall another well known property of big-O notation: $O(log_2 n)$ means the same as $O(log_3 n)$, and so on. The choice of logarithmic base makes no difference to big-O classification. We simply say that $T_{fastexp}(n)$ is $O(log n)$.

### 3.3 Powers of 2, faster or slower

Here is another exponentiation function, $\text{pow} : \text{int} \rightarrow \text{int}$, whose design is based on the facts that for $n > 1$, if $n$ is even then $2^n = (2^n \div 2)^2$ and if $n$ is odd then $2^n = 2(2^n \div 2)^2$. We give this function a different name, so we can compare it with the previous functions.

```haskell
fun pow 0 = 1
| pow 1 = 2
| pow n = let
  val k = pow(n div 2)
  in
    if n mod 2 = 0 then k*k else 2*k*k
  end
```

We use the local variable $k$ to hold the value returned by the recursive call, and this variable gets used twice (no matter which branch gets chosen). This turns out to be crucial in improving efficiency!

Again it is easy to prove by induction on $n$ that for all $n \geq 0$, $\text{pow } n = 2^n$. Indeed this function does compute powers of 2. How about its running time, when applied to a non-negative integer?

In each recursive call, the argument gets halved. So we should expect logarithmic running time. Our recurrence analysis confirms this. Let $W_{pow}(n)$ be the runtime of $\text{pow } n$, for $n \geq 0$. The function definition tells us that there are constants $c_0, c_1, c_2$ such that:

- $W_{pow}(0) = c_0$
- $W_{pow}(1) = c_1$
- $W_{pow}(n) = c_2 + W_{pow}(n \div 2)$ if $n > 1$

This is essentially the same recurrence as the one for $W_{fastexp}$, so the runtime of $\text{pow } n$ is $O(log n)$, the same as for $\text{fastexp}(n)$, asymptotically.
The use of a local variable in the above function definition, to save and re-use the value returned by the recursive call, is crucial for efficiency. In contrast, here is a bad version `badpow : int -> int` that makes redundant recursive calls. Compare the code with that of `pow`.

```ml
fun badpow 0 = 1
| badpow 1 = 2
| badpow n = let
  val k2 = badpow(n div 2) * badpow(n div 2)
  in
  if n mod 2 = 0 then k2 else 2*k2
end
```

Let $W_{\text{badpow}}(n)$ be the runtime of $\text{badpow } n$, for $n \geq 0$. Then (again, from the function definition) we can derive the recurrence

\[
W_{\text{badpow}}(0) = 1 \\
W_{\text{badpow}}(1) = 1 \\
W_{\text{badpow}}(n) = 1 + 2 \times W_{\text{badpow}}(n \div 2) \text{ if } n > 1
\]

If $n$ is a power of 2, say $n = 2^k$, we have $W_{\text{badpow}}(2^k) = 2 \times W_{\text{badpow}}(2^{k-1}) + 1$. Expanding out a few examples, we see that

\[
W_{\text{badpow}}(2^0) = 1 \\
W_{\text{badpow}}(2^1) = 1 + 2W_{\text{badpow}}(2^0) = 1 + 2 = 3 \\
W_{\text{badpow}}(2^2) = 1 + 2W_{\text{badpow}}(2^1) = 1 + 2 + 4 = 7
\]

These examples suggest that for $k \geq 0$, $W_{\text{badpow}}(2^k) = 2^{k+1} - 1$. Indeed this can be shown by induction on $k$. So $W_{\text{badpow}}(2^k)$ is $O(2^k)$, and it can further be shown that for generally $W_{\text{badpow}}(n)$ is $O(n)$, so `badpow` has linear runtime!

Clearly, we should prefer `pow`, with logarithmic running time, over `badpow`, with linear runtime.

This simple example shows that attention to detail and careful design can improve efficiency.
3.4 General exponentiation

We can easily derive a function for computing \( b^n \), where \( n \geq 0 \) and \( b \) is an integer. The main idea is that when \( n \) is even and greater than 2, \( b^n = (b^2)^{n \div 2} \). We didn’t take advantage of this kind of math fact earlier, because we were fixated on computing powers of 2. In tackling the more general task we will exploit this idea to develop a faster algorithm.

\[
\begin{align*}
(* \text{gexp : int * int -> int} *) \\
\text{fun gexp (b, 0) = 1} \\
\mid \text{gexp (b, 1) = b} \\
\mid \text{gexp (b, n) = 1 + gexp (b*b, n div 2)} \\
\quad \text{let} \\
\quad \quad \text{val k = gexp (b*b, n div 2)} \\
\quad \quad \text{in} \\
\quad \quad \text{if n mod 2 = 0 then k else b*k} \\
\quad \text{end}
\end{align*}
\]

It is (again!) easy to prove by induction on \( n \) that for all \( b \) and all \( n \geq 0 \), \( \text{gexp}(b, n) = b^n \). The runtime of \( \text{gexp}(b, n) \) is \( O(\log n) \). We can easily adapt the analysis for \( \text{pow} \) to show this.

3.5 Fibonacci numbers

Here is a recursive ML function \( \text{fib : int -> int} \) for computing Fibonacci numbers. For \( n \geq 0 \) the \( n \)th Fibonacci number is the value of \( \text{fib} n \).

\[
\begin{align*}
\text{fun fib 0 = 1} \\
\mid \text{fib 1 = 1} \\
\mid \text{fib n = fib(n-1) + fib(n-2)}
\end{align*}
\]

This ML definition looks just like the usual way mathematicians define the Fibonacci series, as a recurrence with base cases for 0 and 1.

If we use this function in the ML interpreter window we will see that \( \text{fib} 42 \) takes a very long time to return its result; and \( \text{fib} 43 \) raises the \text{Overflow} exception, because the 43rd Fibonacci number is too large.

Let \( W_{\text{fib}}(n) \) be the running time for \( \text{fib}(n) \). Then, choosing the relevant constants to be 1, we obtain the recurrence relation

\[
\begin{align*}
W_{\text{fib}}(0) &= 1 \\
W_{\text{fib}}(1) &= 1 \\
W_{\text{fib}}(n) &= 1 + W_{\text{fib}}(n-1) + W_{\text{fib}}(n-2) \quad \text{for} \ n > 1
\end{align*}
\]
While this recurrence doesn’t seem easily solvable (at least, not explicitly), it is obvious that $\text{fib}(n) \leq W_{\text{fib}}(n)$ for all $n \geq 0$. And mathematicians have proven\(^1\) that $\text{fib}(n)$ is exponential in $n$, so $W_{\text{fib}}$ has at least exponential running time. No wonder $\text{fib}$ 42 is so slow! It can be shown that $W_{\text{fib}}(n)$ is actually $O(\text{fib}(n))$, so $\text{fib}$ indeed has exponential running time.

We can speed up the code by computing two Fibonacci numbers in each iteration; the function $\text{fastfib} : \text{int} \to \text{int}$ does this, using a locally declared helper function $\text{loop} : \text{int} \times \text{int} \times \text{int} \to \text{int}$ whose second and third arguments accumulate the successive Fibonacci numbers:

\[
\text{fun fastfib n = }
\text{let }
\text{fun loop(i, a, b) = if i=0 then a else loop(i-1, b, a+b)}
\text{in }
\text{loop(n, 1, 1)}
\text{end}
\]

Let $W_{\text{loop}}(n)$ be the running time for $\text{loop}(n, a, b)$, when $n \geq 0$. (Clearly the running time does not depend on the values of $a$ and $b$.) We have, from the function definition, that there are constants $c_0, c_1$ such that

\[
W_{\text{loop}}(0) = c_0 \\
W_{\text{loop}}(i) = c_1 + W_{\text{loop}}(i - 1) \text{ for } i > 0
\]

Hence $W_{\text{loop}}(n)$ is $O(n)$. And therefore so is the running time of $\text{fastfib} \ n$.

Actually, $\text{fastfib}$ 42 returns the result very quickly, but $\text{fastfib}$ 43 raises the Overflow exception because the 43rd Fibonacci number is too large. Note that the functions $\text{fib}$ and $\text{fastfib}$ are extensionally equivalent, even though their runtimes differ significantly.

### 3.6 List reversal

In ML the built-in list append operator @ is an infix operator, and evaluates its arguments from left to right. For list expressions $E_1$ and $E_2$, evaluating $E_1@E_2$ takes the sum of the times to evaluate $E_1$ and $E_2$, plus time proportional to the length of (the value of) $E_1$. Even for list values $L_1$ and $L_2$, which require no further evaluation, it still takes time proportional to the length of $L_1$ to evaluate the append expression $L_1@L_2$. In contrast, a cons

\(^1\)We also included a proof of this result in the lecture notes for the previous week!
expression \( x :: E \) takes time proportional to the sum of the times to evaluate \( x \) and \( E \), plus a constant for the final :: operation. So the work to evaluate \( x :: L \) when \( x \) and \( L \) are values is constant-time.\(^2\) Sometimes it can be very inefficient to use \( @ \) to build lists, especially when there is an alternative way to achieve the same results by using :: instead. Here is an example designed to illustrate this idea.

Here is the obvious list reversal function, which uses append.

\[
\text{fun rev [ ] = [ ]}
\]
\[
| \text{rev (x::L) = rev(L) @ [x]};
\]

Of course, the reverse list of \( L \) is a list consisting of the same items as \( L \) but in the opposite order. For instance \( \text{rev [1,2,3]} = [3,2,1] \).

The running time for \( \text{rev}(L) \), when \( L \) is a list value, depends only on the length of \( L \). Let \( W_{\text{rev}}(n) \) be the runtime for \( \text{rev}(L) \) on list values of length \( n \). From the function definition we can see that

\[
\begin{align*}
W_{\text{rev}}(0) &= c_0 \\
W_{\text{rev}}(n) &= W_{\text{rev}}(n - 1) + c_1 + n
\end{align*}
\]

for some constants \( c_0 \) and \( c_1 \). So, expanding out a few cases, we get

\[
\begin{align*}
W_{\text{rev}}(1) &= c_0 + c_1 + 1 \\
W_{\text{rev}}(2) &= (c_0 + c_1 + 1) + c_1 + 2 \\
&= c_0 + 2 \ast c_1 + (1 + 2) \\
W_{\text{rev}}(3) &= W_{\text{rev}}(2) + c_1 + 3 \\
&= c_0 + 3 \ast c_1 + (1 + 2 + 3)
\end{align*}
\]

Note that the sum of the first \( n \) positive integers is equal to \( \frac{1}{2}n(n+1) \). Guided by these examples, it is easy to prove, by induction on \( n \), that for all \( n \geq 0 \),

\[
W_{\text{rev}}(n) = c_0 + n \ast c_1 + \frac{1}{2}n(n+1).
\]

Hence \( W_{\text{rev}}(n) \) is quadratic in \( n \). So the runtime of \( \text{rev}(L) \) is quadratic in the length of \( L \).

\(^2\)The ML implementation of \( @ \) uses :: repeatedly, to prepend the items on \( L_1 \) onto the front of \( L_2 \). That's why the time to evaluate \( L_1 @ L_2 \) is proportional to the length of \( L_1 \).
Faster reversal

There is a fairly obvious way to do reversal more efficiently, by introducing an extra list argument to the function and consing onto the front of it. There’s no need to do any @ anywhere here!

\[
\begin{align*}
(* \text{revver} : & \text{int list } \times \text{int list } \rightarrow \text{int list } *) \\
\text{fun revver([ ]}, & \ A) = \ A \\
& | \ \text{revver}(\text{x::L}, \ A) = \text{revver}(\text{L, x::A}); \\
\end{align*}
\]

\[
\begin{align*}
(* \text{Rev} : & \text{int list } \rightarrow \text{int list } *) \\
\text{fun Rev} \ L & = \text{revver}(\text{L, [ ]}); \\
\end{align*}
\]

It is easy to show (by list induction on \( L \)) that for all integer lists \( L, A \), \( \text{revver}(L, A) = (\text{rev} \ L) \times A \). Hence, \( \text{Rev} \ L = \text{rev} \ L \), for all \( L \).

The runtime of \( \text{revver}(L, A) \) is linear in length of \( L \). (Show why, using the techniques from above.)

The runtime of \( \text{Rev}(L) \) is linear in the length of \( L \).

Warning

Introducing an accumulator to improve efficiency is a widely useful technique. The fastfib and revver examples serve to illustrate the idea. As we will see later, depending on the setting, you may want to choose the accumulator to be a list, an integer, a function, or a value of some other type. But this technique is not a panacea: the trick doesn’t always truly improve runtime!

Consider:

\[
\begin{align*}
(* \text{exp'} : & \text{int } \times \text{int } \rightarrow \text{int } *) \\
\text{fun exp'} & (n, a) = \text{if } n=0 \text{ then a else exp'}(n-1, 2\times a); \\
\text{fun Exp} \ n & = \text{exp'}(n, 1); \\
\end{align*}
\]

\( \text{exp'} \) is obtained from \( \text{exp} \) by adding an accumulator integer argument, but has the same asymptotic behavior. And \( \text{Exp} \) is extensionally equivalent to \( \text{exp} \) from before. The runtime for \( \text{exp'}(n, a) \) is also \( O(n) \), just like the runtime for \( \text{exp}(n) \).
4 big-O classes

- \(O(1)\), or constant time
- \(O(\log n)\), or logarithmic
- \(O(n)\), or linear
- \(O(n^2)\), or quadratic
- \(O(n^3)\), or cubic
- \(O(2^n), O(3^n), \ldots\) exponentials (each is a different class)

5 Some useful facts

The following facts can help explain common terminology:

- \(O(\log_2 n)\) is the same class of functions as \(O(\log_{10} n)\). In fact the base of the logarithm makes no difference to the class of functions, so we usually just write \(O(\log n)\) and refer to “logarithmic” time.

- A function is called polynomial time if it is \(O(n^k)\) for some \(k \geq 0\).

- A polynomial function with highest power \(k\) is \(O(n^k)\).

- A linear function of \(n\) has the form \(\alpha n + \beta\), for some constants \(\alpha\) and \(\beta\). Every linear function is \(O(n)\).

- Every function that is \(O(n^2)\) is also \(O(n^3)\), but the converse fails.

- Every function that is \(O(2^n)\) is also \(O(3^n)\), but the converse fails.

- A function of \(n\) is said to be exponential time if it is \(O(k^n)\) for some constant \(k\).

- Every polynomial time function is also exponential time. (We’re not going to make any claims about the converse!)

- If \(f(n)\) is \(O(g(n))\), then \(O(f(n) + g(n))\) is the same as \(O(g(n))\).
6 Common recurrences

It’s common to use sloppy notation and write something like $O(g(n))$ in a recurrence, in a place where an actual function of $n$ is intended. For example if we write $W(n) = O(n) + W(n - 1)$ we mean that there is some function $f(n) \in O(n)$ such that $W(n) = f(n) + W(n - 1)$. It doesn’t make any difference, asymptotically. For any non-zero linear function $f$ the solution to this recurrence is going to be $O(n^2)$.

We give the clause for $n > 0$. In each case $c$ and/or $k$ are constants. We mention in each case the most informative time-complexity class of the solution to the recurrence relation.

- $T(n) = T(n \text{ div } 2) + c$, or $T(n) = T(n \text{ div } 2) + O(1)$
  $T(n)$ is $O(\log n)$

- $T(n) = T(n \text{ div } 4) + c$, also $O(\log n)$.

- $T(n) = T(n - 1) + c$, or $T(n) = T(n - 1) + O(1)$
  $T(n)$ is $O(n)$

- $T(n) = 2 \times T(n \text{ div } 2) + c$, or $T(n) = 2 \times T(n \text{ div } 2) + O(1)$
  $T(n)$ is $O(n)$

- $T(n) = T(n - 1) + c \times n$, or $T(n) = T(n - 1) + O(n)$
  $T(n)$ is $O(n^2)$

- $T(n) = 2 \times T(n \text{ div } 2) + c \times n$, or $T(n) = 2 \times T(n \text{ div } 2) + O(n)$
  $T(n)$ is $O(n \log n)$

- $T(n) = k \times T(n - 1) + c$, $k > 1$
  $T(n)$ is $O(k^n)$
  This is also the case for $T(n) = k \times T(n - 1) + O(1)$. 
7 Guessing or estimating solutions

Often it is easy to expand out a few examples and look for a pattern from which we can guess a solution. Here we sketch how to justify some of the above facts about common recurrences. Since we are only sketching the ideas we won’t provide completely rigorous inductive proofs.

- \( T(n) = T(n \text{ div } 2) + c \)
  
  For \( n = 2^m \) with \( m > 0 \) we have
  \[
  T(2^m) = T(2^{m-1}) + c = T(2^{m-2}) + c + c
  \]
  
  and it looks like \( T(2^m) = T(2^{m-k}) + kc \), for \( 0 \leq k \leq m \). In particular, \( T(2^m) = T(0) + mc \). This suggests that \( T(2^m) \) is \( O(m) \). Extrapolating to all values of \( n \), we expect that \( T(n) \) is \( O(\log n) \).

- \( T(n) = 2 \ast T(n \text{ div } 2) + c \)
  
  For \( n = 2^m \) with \( m > 0 \) we have
  \[
  T(2^m) = 2T(2^{m-1}) + c = 2(2T(2^{m-2}) + c) + c = 2^2T(2^{m-2}) + (2 + 1)c
  \]
  
  and it looks like
  \[
  T(2^m) = 2^kT(2^{m-k}) + (2^{k-1} + \cdots + 2 + 1)c
  \]
  
  for \( 0 \leq k \leq m \). In particular, putting \( k = m \),
  \[
  T(2^m) = 2^mT(1) + (2^{m-1} + \cdots + 2 + 1)c.
  \]

Note that \( 2^{m-1} + \cdots + 2 + 1 = 2^m - 1 \) is \( O(2^m) \). So this analysis suggests that \( T(2^m) \) is \( O(2^m) \). Extrapolating to all values of \( n \), we expect that \( T(n) \) is \( O(n) \). When extrapolating like this we are usually appealing to the fact that when \( 2^k \leq n < 2^{k+1} \) we have \( T(2^k) \leq T(n) \leq T(2^{k+1}) \), so if \( T(2^k) \) is approximately \( ck \) we get \( ck \leq T(n) \leq c(k+1) \), from which we see that \( T(n) \leq c\log_2 n + c \). Hence \( T(n) \) is \( O(\log n) \).

This result is consistent with the table of standard recurrences given earlier.
8 Going forward

- Get used to deriving recurrences for your recursive function designs and choosing more efficient designs when available.

- Be aware of how you could solve recurrences inductively, either exactly or asymptotically. Practice on examples.

- Learn to recognize commonly occurring recurrences: this can save you a lot of time!

- You won’t need to work extensively with the internal details of big-O notation, although we have given you some insights that show the way.
9 Self-test 5

1. Explore the behavior of the following ML functions to see the growth rates of various recurrences:

fun A(n) = if n=0 then 1 else A(n div 4) + 1
fun B(n) = if n=0 then 1 else 2*B(n div 4) + 1
fun C(n) = if n=0 then 1 else 3*C(n div 4) + 1
fun D(n) = if n=0 then 1 else 4*D(n div 4) + 1
fun E(n) = if n=0 then 1 else 5*E(n div 4) + 1

What can you say about the asymptotic runtimes for these functions?

2. Let \( W(n) \) be defined by the following recurrence, in which \( c_0 \) and \( c_1 \) are positive constants:

\[
W(0) = c_0 \\
W(n) = c_1 n + W(n-1), \text{ for } n > 0,
\]

Prove by induction that for all \( n \geq 0 \), \( W(n) = c_0 + \frac{1}{2} n(n+1)c_1 \). Explain why this implies that \( W(n) = O(n^2) \).

3. Using the definition of “\( f = O(g) \)”, show that when \( c_0 \) and \( c_1 \) are integers and \( c_1 \neq 0 \), the function \( f(n) = c_0 + nc_1 \) is not \( O(1) \).

4. Consider the function \( W_1 \) given by the following recurrence:

\[
W_1(0) = 1 \\
W_1(n) = 3W_1(n-1) + 1 \text{ for } n > 0
\]

(a) Draw a table showing the values of \( W_1(n) \) for \( n = 0, 1, 2, 3, 4 \).
(b) Augment your table to show the values of \( W_1(k+1) - W_1(k) \) for \( k = 0, 1, 2, 3 \).
(c) Prove, by induction on \( n \), that for all \( n \geq 0 \), \( W_1(n) = \frac{1}{2}(3^{n+1} - 1) \).
(d) Deduce that \( W_1(n) = O(3^n) \).

5. Now consider the function \( W_2 \) given by the following recurrence, on non-negative integer arguments:

\[
W_2(0) = 1 \\
W_2(1) = 1 \\
W_2(n) = 3W_2(n-2) + 1 \text{ for } n > 1
\]

18
(a) Draw a table showing the values of $W_2(n)$ for $n = 0, 1, 2, 3, \ldots, 8$.

(b) Prove by induction on $n$ that for all $n \geq 0$, $W_2(n) = W_1(n \text{ div } 2)$, where \text{div} is integer division, so that $3 \text{ div } 2 = 1$ and so on.

(c) Deduce that $W_2(n)$ is $O(3^n \text{ div } 2)$.

6. Is $O(3^n \text{ div } 2) \subseteq O(3^n)$? Is $O(3^n) \subseteq O(3^n \text{ div } 2)$? Say why.

7. Suppose we are given the following property of the function \texttt{fastexp}:

For all values $n \geq 1$,

$$\texttt{fastexp}(n) \mapsto (c_0 \log_2(n) + c_1) \cdot k,$$

where $k$ is the value of $2^n$, $c_0$ and $c_1$ are unspecified constants, $c_0 \neq 0$, and $\log_2(n)$ is the (integer) logarithm base 2 of $n$.

(a) How many steps does it take to evaluate the expression

$$\texttt{fastexp}(\texttt{fastexp}(n)),$$

when $n \geq 1$.

(b) Give a big-O classification for the work of this expression.

(c) From the above property of \texttt{fastexp}, does it follow that $W_{\text{fastexp}}(n)$ is $O(\log_2(n))$? How about $O(n)$?

• Why did we not include $n = 0$ in the statement of the above property?

8. Consider the following function definitions:

```plaintext
fun upto (i:int, j:int) : int list =
  if i>j then [] else i :: upto (i+1, j);

fun sum [ ] = 0
  | sum (x::L) = x + (sum L)
```

The work to evaluate \texttt{upto(1,n)} for an integer value $n$ is $O(n)$. The work to evaluate $A @ B$ when $A$ and $B$ are list values is $O(\text{length}(A))$. For each of the following expressions, give a big-O estimate of the work to evaluate the expression. Be as accurate as you can.
Define an ML function $\text{foo} : \text{int} \rightarrow \text{int}$ such that for all $n \geq 0$, $\text{foo}(n)$ returns the same integer result as does each of the expressions (a), (b), (c) above, but the work to evaluate $\text{foo}(n)$ is $O(1)$.

9. The \textit{integer square root} of a non-negative integer $n$ is defined to be the largest non-negative integer $y$ such that $y^2 \leq n$. For every $n \geq 0$ such a $y$ exists, because the square numbers $0^2, 1^2, 2^2, 3^2 \ldots$ form a strictly increasing sequence of non-negative integers and must exceed $n$ after some number of steps. We can also describe the integer square root of $n$ as the unique non-negative integer $y$ such that $y^2 \leq n < (y + 1)^2$.

Here are some functions for computing integer square roots. They both REQUIRE a non-negative argument.

\begin{verbatim}
fun sqrt1 (n:int) : int = 
  if n=0 then 0 else 
  let 
    val r = sqrt1 (n-1) + 1 
  in 
  if n < r*r then r - 1 else r 
end

fun sqrt2 (n : int) : int = 
  if n=0 then 0 else 
  let 
    val r = 2 * sqrt2 (n div 4) + 1 
  in 
  if n < r*r then r - 1 else r 
end
\end{verbatim}

(a) Sketch a proof (using induction) that for all $n \geq 0$, $\text{sqrt1 } n$ evaluates to the integer square root of $n$.

HINT: Let $y > 0$ and $y^2 \leq n - 1 \leq (y + 1)^2$. Show that

- If $n < (y + 1)^2$, it follows that $y^2 \leq n < (y + 1)^2$. 

20
(b) Sketch a proof (using induction) that for all \( n \geq 0 \),
\( \text{sqrt2} \ n \) evaluates to the integer square root of \( n \).
HINT: Let \( y > 0 \) and \( y^2 \leq n \div 4 \leq (y + 1)^2 \). Show that
- If \( n < (2y + 1)^2 \), it follows that \( (2y)^2 \leq n < (2y + 1)^2 \).
- If \( n \geq (2y + 1)^2 \) it follows that \( (2y + 1)^2 \leq n < (2y + 2)^2 \).
Remember that \( n = 4(n \div 4) + (n \mod 4) \) and \( 0 \leq n \mod 4 < 4 \).

10. Again referring to the two functions for computing integer square roots, consider how efficient they are.
- What pattern of recursive calls occurs when evaluating \( \text{sqrt1} \ 100 \)?
- What pattern of recursive calls occurs when evaluating \( \text{sqrt2} \ 100 \)?
- How many squaring operations \( \ast \) get done in the evaluation of \( \text{sqrt1} \ 100 \)?
- How many squaring operations are done for \( \text{sqrt2} \ 100 \)?
- What is the asymptotic work to evaluate \( \text{sqrt1} \ n \), for \( n \geq 0 \)?
- What is the asymptotic work to evaluate \( \text{sqrt2} \ n \), for \( n \geq 0 \)?

11. Refer back to the \text{fastfib} function that uses a local helper function \text{loop}. Prove that \( \text{fib} = \text{fastfib} \). You will need to state and prove a suitable specification for \text{loop}.

12. Refer to the \text{rev} and \text{revver} definitions given earlier.
   (a) Prove that for all \( L \) and \( A \) of the same list type,
   \[
   \text{revver}(L, A) = (\text{rev} \ L)@A.
   \]
   (b) Show that the work to evaluate \text{revver}(L, A), when \( L \) and \( A \) are lists (values) of the same type, is linear in the length of \( L \) (so independent of the length of \( A \)). What is the work to evaluate \( (\text{rev} \ L)@A \)?