15–150: Principles of Functional Programming

Work Analysis

Michael Erdmann*
Spring 2026

1 Introduction

Today's lecture introduces techniques for analyzing the runtime of functional programs. You should already be familiar with the basic concepts, but we give a brief recap of the main ideas.

2 Main Points

- We show how to obtain a *recurrence relation* for the runtime of an SML 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 introducing an "accumulator", or by computing extra information.
- We give a first example for a span analysis using recurrences.

3 Asymptotic Analysis

We will focus on *asymptotic analysis* of programs. 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 consing 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.

*Adapted from a document by Stephen Brookes.

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 probably prefer a running time of $50000n^3$ to a running time of 2^n, when n is large, 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 `int -> int` we say that "f is `O`(g)" if there is a constant c and an integer N such that for all $n \geq N$, $|f(n)| \leq c|g(n)|$.

When $f(n)$ 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 "for all $n \geq N$, $f(n) \leq c * g(n)$".

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 $f(n) = 30n^2 + 4000n + 1 \ldots$").

We may take advantage of well known results about big-O notation, for instance the fact that "constants don't matter". On page 7, we summarize some key results.

4 Examples

In these examples, we sometimes omit explicit type annotations and almost always omit `REQUIRES` and `ENSURES` clauses, because we want to focus on runtime analysis. In homework and lab, please continue to include this information for any functions that you write.

4.1 Powers of 2

The SML function `exp` given next calculates powers of 2.

```
(* exp : int -> int *)
fun exp (0:int):int = 1
  | exp (n:int):int = 2 * exp (n-1)
```

It is easy to prove by induction that for all $n \geq 0$, `exp` n evaluates to 2^n.

Let $W_{\texttt{exp}}(n)$ be the running time (or "work") of `exp` n, for $n \geq 0$. We assume (as usual) that arithmetic and Boolean operations take constant time. From the structure of the function definition, we see that there are positive constants c_0, c_1 such that

$$\begin{aligned} W_{\texttt{exp}}(0) &= c_0 \\ W_{\texttt{exp}}(n) &= c_1 + W_{\texttt{exp}}(n-1), \quad \text{for } n > 0. \end{aligned}$$

Using this recurrence relation, we may prove by induction on n, that for all $n \geq 0$, $W_{\texttt{exp}}(n) = n * c_1 + c_0$:

- Base Case: $n = 0$. Need to show $W_{\texttt{exp}}(0) = c_0$.
  That is given explicitly by the first equation for $W_{\texttt{exp}}$.
- Induction Step: Step from n to $n + 1$, with $n \geq 0$.

– Induction Hypothesis: $W_{\texttt{exp}}(n) = n * c_1 + c_0$.
– Need to Show: $W_{\texttt{exp}}(n+1) = (n+1) * c_1 + c_0$.
– Showing:

$$\begin{array}{rcll} W_{\texttt{exp}}(n+1) & = & c_1 + W_{\texttt{exp}}((n+1)-1) & \text{[by second recurrence equation]} \\ & = & c_1 + W_{\texttt{exp}}(n) & \\ & = & c_1 + n * c_1 + c_0 & \text{[by IH]} \\ & = & (n+1) * c_1 + c_0. & \end{array}$$

□

$W_{\texttt{exp}}(n) = n * c_1 + c_0$, is called a *closed form* solution of the recurrence relation. This closed form tells us that $W_{\texttt{exp}}(n)$ is *linear* in n, i.e., that $W_{\texttt{exp}}(n)$ is $\texttt{O}(n)$, as follows:

We know that there are (positive) constants c_0 and c_1 such that $W_{\texttt{exp}}(n) = n * c_1 + c_0$, for all $n \geq 0$. Pick c to be $c_1 + 1$ and let $N \geq c_0$. Then for all $n \geq N$ we have

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

Thus, according to the definition of big-O, $W_{\texttt{exp}}(n)$ is $\texttt{O}(n)$.

Actually, it can be convenient to make a simplifying assumption about these "unknown" constants. For instance, for any positive constants c_0 and c_1, the function $f(n) = n * \texttt{c}_0 + \texttt{c}_1$ is $\texttt{O}(n)$. The choice of constants makes no difference to this fact. So we could have made an arbitrary decision to choose $\texttt{c}_0 = \texttt{c}_1 = 1$ and taken the recurrence defining $W_{\texttt{exp}}$ to be

$$\begin{array}{rcll} W_{\texttt{exp}}(0) & = & 1 & \\ W_{\texttt{exp}}(n) & = & 1 + W_{\texttt{exp}}(n-1), & \text{for } n > 0. \end{array}$$

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

4.2 Powers of 2, faster

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

```
fun square (x:int):int = x*x

(* fastexp : int -> int *)
fun fastexp (0:int):int = 1
  | fastexp (n:int):int =
     (case (n mod 2) of
         0 => square (fastexp (n div 2))
       | _ => 2 * fastexp (n-1))
```

Again it is easy to prove that for all $n \geq 0$, `fastexp` n evaluates to 2^n.

Now let $W_{\texttt{fastexp}}(n)$ be the runtime of `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:

$$\begin{array}{rcll} W_{\texttt{fastexp}}(0) & = & k_0 & \\ W_{\texttt{fastexp}}(n) & = & k_1 + W_{\texttt{fastexp}}(n \texttt{ div } 2) & \text{if } n > 0 \text{ and } n \text{ even} \\ W_{\texttt{fastexp}}(n) & = & k_2 + W_{\texttt{fastexp}}(n-1) & \text{if } n > 0 \text{ and } n \text{ odd} \end{array}$$

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

$$\begin{aligned} W_{\texttt{fastexp}}(0) &= k_0 \\ W_{\texttt{fastexp}}(n) &= k_1 + W_{\texttt{fastexp}}(n \texttt{ div } 2) && \text{if } n > 0 \text{ and } n \text{ even} \\ W_{\texttt{fastexp}}(n) &= k_2 + k_1 + W_{\texttt{fastexp}}(n \texttt{ div } 2) && \text{if } n > 0 \text{ and } n \text{ odd.} \end{aligned}$$

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

$$\begin{aligned} T_{\texttt{fastexp}}(0) &= 1 \\ T_{\texttt{fastexp}}(1) &= 1 \\ T_{\texttt{fastexp}}(n) &= 1 + T_{\texttt{fastexp}}(n \texttt{ div } 2) && \text{for } n > 1. \end{aligned}$$

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

Indeed this recurrence for $T_{\texttt{fastexp}}$ is *exactly the same recursive pattern* as one uses to define the logarithm function `log : int -> int`; this function computes logarithms in base 2. So we can get a closed form for $T_{\texttt{fastexp}}(n)$: For all $n \geq 1$, $T_{\texttt{fastexp}}(n) = \texttt{log}_2(n) + 1$. Recall that $\log_2 n$ is the largest non-negative integer k such that $2^k \leq n$.

This doesn't imply that $W_{\texttt{fastexp}}(n)$ is also equal to $\texttt{log}_2(n)$ — it couldn't be, because its recurrence relation mentions k_0, k_1, k_2. But we said that $W_{\texttt{fastexp}}$ and $T_{\texttt{fastexp}}$ have the same asymptotic behavior; $W_{\texttt{fastexp}}(n)$ is in the same $\texttt{O}$-class as $T_{\texttt{fastexp}}(n)$. Hence $W_{\texttt{fastexp}}(n)$ is $\texttt{O}(\log_2 n)$.

Recall as well that for any two bases a and b, $\log_b n = c \log_a n$, with constant $c = \log_b a$. Thus $\texttt{O}(\log_b n)$ is the same class as $\texttt{O}(\log_a n)$. The choice of logarithmic base makes no difference to big-O classification. We simply say that $T_{\texttt{fastexp}}(n)$ is $\texttt{O}(\log n)$.

4.3 Powers of 2, not so fast

Here is a rewrite of the fast exponentiation function that does *not* give a speedup.

```
(* badexp : int -> int *)
fun badexp (0:int):int = 1
  | badexp (n:int):int =
     (case (n mod 2) of
         0 => (badexp (n div 2)) * (badexp (n div 2))
       | _ => 2 * badexp (n-1))
```

Notice that `badexp` makes the same recursive call twice now, rather than making it once and squaring the result as `fastexp` did. Consequently, although the functions `fastexp` and `badexp` are extensionally equivalent, they have different running times, as we see next.

Let $W_{\texttt{badexp}}(n)$ be the runtime of `badexp` n, for $n \geq 0$. Then (again, from the function definition) we can, for some constants c_0, c_1, and c_0, derive the recurrence

$$\begin{aligned} W_{\texttt{badexp}}(0) &= c_0 \\ W_{\texttt{badexp}}(1) &= c_1 \\ W_{\texttt{badexp}}(n) &= c_2 + 2 * W_{\texttt{badexp}}(n \texttt{ div } 2) && \text{for } n > 1. \end{aligned}$$

One can show by strong induction that $W_{\texttt{badexp}}(n)$ is $\texttt{O}(n)$, so `badexp` has *linear* runtime, just as did the first exponentiation function, `exp`.

We probably prefer `fastexp`, with logarithmic running time, over `badexp`, with linear runtime.

4.4 Fibonacci numbers

Here is an SML implementation that corresponds to the usual mathematical presentation of the Fibonacci series. For $n \geq 0$ we represent the n^{th} Fibonacci number as the value of `fib` n.

```
(* fib : int -> int *)
fun fib 0 = 1
 |  fib 1 = 1
 |  fib n = fib(n-1) + fib(n-2)
```

If we use this function in the SML interpreter window we will see that `fib` 42 takes a very long time to return its result.

(Aside: `fib` 44 raises the `Overflow` exception, because the 44^{th} Fibonacci number is too large.)

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

$$\begin{aligned} W_{\texttt{fib}}(0) &= 1 \\ W_{\texttt{fib}}(1) &= 1 \\ W_{\texttt{fib}}(n) &= 1 + W_{\texttt{fib}}(n-1) + W_{\texttt{fib}}(n-2) \quad \text{for } n > 1. \end{aligned}$$

This tells us that $\texttt{fib}(n) \leq W_{\texttt{fib}}(n)$ for all $n \geq 0$. Since Fibonacci numbers grow exponentially fast, this tells us that $W_{\texttt{fib}}$ has *at least* exponential running time. No wonder `fib` 42 is so slow! It can be shown that $W_{\texttt{fib}}(n)$ is actually $\texttt{O}(\texttt{fib}(n))$, so `fib` has exponential running time.

We can speed up the function by returning two Fibonacci numbers instead of one:

```
(* ffib: int -> int*int
   REQUIRES: n >= 0
   ENSURES: ffib(n) ==> (f_n, f_{n-1}), the nth and {n-1}st Fibonacci numbers,
                                        where we define f_{-1} = 0.
*)
fun ffib (0:int):int*int = (1, 0)
  | ffib (n:int):int*int =
                let
                  val (f1, f2) : int*int = ffib (n-1)
                in
                  (f1 + f2, f1)
                end
```

Let $W_{\texttt{ffib}}(n)$ be the running time for `ffib`(n), when $n \geq 0$. We have, from the function definition, that there are constants c_0, c_1 such that

$$\begin{aligned} W_{\texttt{ffib}}(0) &= c_0 \\ W_{\texttt{ffib}}(n) &= c_1 + W_{\texttt{ffib}}(n-1) \quad \text{for } n > 0. \end{aligned}$$

Hence $W_{\texttt{ffib}}(n)$ is $\texttt{O}(n)$.

4.5 List reversal

The list append operation $L_1 \texttt{@} L_2$ takes time proportional to the length of L_1. The list cons construction $x::L$ takes constant time.

Recall this list reversal function, based on append:

```
(* rev : int list -> int list *)
fun rev [ ] = [ ]
  | rev (x::L) = rev(L) @ [x]
```

Let $W_{\texttt{rev}}(n)$ be the runtime for `rev(L)` on lists of length n. From the function definition we can see that

$$\begin{aligned} W_{\texttt{rev}}(0) &= c_0 \\ W_{\texttt{rev}}(n) &= W_{\texttt{rev}}(n-1) + c_1 + c_2 * n \end{aligned}$$

for some constants c_0, c_1, and c_2. So, expanding out a few cases, we get

$$\begin{aligned} W_{\texttt{rev}}(1) &= c_0 + c_1 + c_2 \\ W_{\texttt{rev}}(2) &= (c_0 + c_1 + c_2) + c_1 + c_2 * 2 \\ &= c_0 + 2 * c_1 + (1+2) * c_2 \\ W_{\texttt{rev}}(3) &= W_{\texttt{rev}}(2) + c_1 + c_2 * 3 \\ &= c_0 + 3 * c_1 + (1+2+3) * c_2. \end{aligned}$$

(**Terminology**: The phrase "unrolling the recurrence" refers to this process of expanding out terms repeatedly, to detect a pattern for the recurrence solution.)

Recall that the sum of the first n positive integers is equal to $\frac{1}{2}n(n+1)$. Indeed, one my prove by induction on n, that for all $n \geq 0$,

$$W_{\texttt{rev}}(n) = c_0 + n * c_1 + \frac{1}{2}n(n+1) * c_2.$$

Hence $W_{\texttt{rev}}(n)$ is quadratic in n, i.e., $O(n^2)$. So the runtime of `rev(L)` is quadratic in the length of `L`.

4.6 Faster reversal

As we saw in Lecture 4, sometimes one may improve efficiency by writing a tail-recursive function, using an extra argument to the function to "accumulate" or build up the final result.

For list reversal we wrote:

```
(* trev : int list * int list -> int list *)
fun trev([ ], acc) = acc
 |  trev(x::L, acc) = trev(L, x::acc)
```

One implements the reversal function of type `int list -> int list` by calling the fast tail-recursive function:

```
(* Rev : int list -> int list *)
fun Rev L = trev (L, [ ])
```

Let $W_{\texttt{trev}}(n, m)$ be the runtime for `trev(L,acc)` when `L` has length n and `acc` has length m. From the function definition we can see that

$$\begin{aligned} W_{\texttt{trev}}(0, m) &= c_0, \quad \text{for all } m \\ W_{\texttt{trev}}(n, m) &= W_{\texttt{trev}}(n-1, m+1) + c_1, \quad \text{for } n > 0 \text{ and for all } m, \end{aligned}$$

for some constants c_0 and c_1. Thus $W_{\texttt{trev}}$ is $O(n)$, meaning that both `trev` and `Rev` have runtimes linear in the length of the list being reversed.

5 Work and Span

So far we have talked about running time for code evaluated sequentially, on a single processor.

Some data structures, such as trees and sequences (which we discuss in more detail later in the semester), allow parallel evaluation. In general when trying to analyze the performance characteristics of programs it is useful to deal with two concepts: *work*, which reflects the total number of steps or operations needed (and corresponds to sequential running time); and *span*, which gives an upper bound on the running time if an unlimited supply of processors is available and we partition the work among as many processors as we need.

Span is determined by the *data dependencies* in a computation: a step that depends on or uses data from another step in the computation must occur later, namely when the data is available.

For sequential code running on a single processor, work is essentially the same as the "time" to evaluate, as in our examples throughout this lecture.

We will discuss work and span in more detail in a few lectures when we return to parallelism, but see Section 8 for an example.

6 Big-O Classes

- $\texttt{O}(1)$, known as *constant time*
- $\texttt{O}(n)$, known as *linear time*
- $\texttt{O}(n^2)$, known as *quadratic time*
- $\texttt{O}(n^3)$, known as *cubic time*
- $\texttt{O}(\log n)$, known as *logarithmic time*
- $\texttt{O}(n \log n)$
- $\texttt{O}(2^n), \texttt{O}(3^n), \ldots$, different classes of *exponential time*

7 Common Recurrences

Here are some common recurrences and their big-O classes (we show only the recursive clause, for $n > 0$, for each recurrence):

		type of recurrence	big-O class
$T(n)$	=	$c_0 + T(n \texttt{ div } 2)$	$\texttt{O}(\log n)$
$T(n)$	=	$c_0 + T(n-1)$	$\texttt{O}(n)$
$T(n)$	=	$c_0 + 2\,T(n \texttt{ div } 2)$	$\texttt{O}(n)$
$T(n)$	=	$c_0 + c_1 n + T(n \texttt{ div } 2)$	$\texttt{O}(n)$
$T(n)$	=	$c_0 + c_1 n + 2\,T(n \texttt{ div } 2)$	$\texttt{O}(n \log n)$
$T(n)$	=	$c_0 + c_1 n + T(n-1)$	$\texttt{O}(n^2)$
$T(n)$	=	$c_0 + c_1 n + c_2 n^2 + T(n-1)$	$\texttt{O}(n^3)$
$T(n)$	=	$c_0 + 2\,T(n-1)$	$\texttt{O}(2^n)$
$T(n)$	=	$c_0 + c_1 2^n + 2\,T(n-1)$	$\texttt{O}(n\,2^n)$

Practice Problem: Derive the big-O class for each recurrence.

8 A Brief Introduction to Parallelism in Trees

Consider the following tree declaration:

```
datatype tree = Empty | Node of tree * int * tree
```

and the following code for summing the integers in a tree:

```
(* sum : tree -> int
   REQUIRES: true
   ENSURES: sum(T) evaluates to the sum of all the integers in T.
*)
fun sum (Empty : tree) : int = 0
  | sum (Node(l,x,r)) = (sum l) + (sum r) + x
```

Let's analyze the work of `sum`. Given a tree `t`, let n be the number of integers in the tree. If `t` is not `Empty`, then let n_ℓ be the number of integers in `t`'s left subtree and n_r the number of integers in `t`'s right subtree. Note that $n = n_\ell + n_r + 1$. Let $W_{\texttt{sum}}(n)$ denote the work required to evaluate `sum(t)`, with `t` containing n integers. We obtain the following recurrence relation:

$$\begin{aligned} W_{\texttt{sum}}(0) &= c_0 \\ W_{\texttt{sum}}(n) &= c_1 + W_{\texttt{sum}}(n_\ell) + W_{\texttt{sum}}(n_r) \quad \text{when } n > 0, \end{aligned}$$

for some constants c_0 and c_1.

With some experience you will recognize that this means $W_{\texttt{sum}}(n)$ is $O(n)$.

In fact, using induction one can prove that:

$$W_{\texttt{sum}}(n) = c_0 + (c_1 + c_0)n.$$

An opportunity for parallelism: In evaluating the recursive clause for `sum`, one may evaluate the two expressions `(sum l)` and `(sum r)` in parallel since there are no data dependencies between these two expressions. Let $S_{\texttt{sum}}(n)$ denote the time required to evaluate `sum(t)` assuming an unlimited number of processors. One obtains the following recurrence relation:

$$\begin{aligned} S_{\texttt{sum}}(0) &= c_0 \\ S_{\texttt{sum}}(n) &= c_1 + \max\left(S_{\texttt{sum}}(n_\ell), S_{\texttt{sum}}(n_r)\right) \quad \text{when } n > 0, \end{aligned}$$

again for some constants c_0 and c_1 (possibly different than before).

Observe that the "+" we saw in the work analysis becomes a "max" in the span analysis.

Unfortunately, the tree could be very lopsided, for instance with $n_\ell = n - 1$ and $n_r = 0$ at each node. In that case, $S_{\texttt{sum}}(n)$ is again $O(n)$, meaning parallelism doesn't help.

Suppose however that the tree is roughly balanced, meaning that each subtree contains roughly half the remaining integers, and this property holds recursively. The second equation in the span recurrence then becomes:

$$S_{\texttt{sum}}(n) \leq c_1 + \max\left(S_{\texttt{sum}}\left(\left\lfloor \frac{n-1}{2} \right\rfloor\right), S_{\texttt{sum}}\left(\left\lceil \frac{n-1}{2} \right\rceil\right)\right) \leq c_1 + S_{\texttt{sum}}(\lfloor n/2 \rfloor).$$

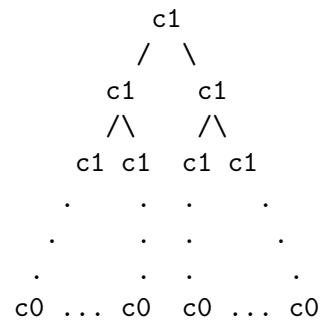
There are roughly $\log_2(n)$ many recursive calls before one reaches an `Empty` tree.

Expanding the recurrence, one therefore sees that

$$S_{\texttt{sum}}(n) \leq c_0 + \sum_{i=1}^{\lfloor \log_2 n \rfloor + 1} c_1 = c_0 + c_1 + c_1 * \lfloor \log_2 n \rfloor,$$

which is $O(\log n)$.

The Tree Method: Sometimes it is helpful in analyzing work and span to depict the evaluation-time recursive calls of a function visually as a tree. For instance, we can visualize the work done by the function `sum` as a certain amount of work done at each node in the tree:



This picture suggests that the work is linear in the number of nodes, i.e., $O(n)$, and that the span is linear in the depth of the tree, i.e., $O(d)$. If the tree is balanced, then d is $O(\log n)$, but if the tree is not balanced, then d could itself be $O(n)$.

(Caution: In the example above, the work at each node is a constant. In more general settings, the work may depend on some other size parameter, perhaps the size of the subtree rooted at that node.)

This method can also be useful for analyzing list functions, where the tree picture now depicts evaluation-time decomposition of the list. For instance, if a function repeatedly splits a list into two equal-sized sublists, then the evaluation time behavior of the function looks like a balanced tree. We will explore that idea next, in the context of sorting.

9 A Peek at Sorting

The code given below implements *insertion sort*.

Be aware that SML contains the following predefined type:

```
datatype order = LESS | EQUAL | GREATER
```

and a comparison function for integers `Int.compare : int * int -> order`.

```
(* ins : int * int list -> int list
   REQUIRES: L is sorted
   ENSURES:  ins(x, L) evaluates to a sorted permutation of x::L
*)
fun ins (x, [ ]) = [x]
  | ins (x, y::L) = (case Int.compare(x, y) of
                        GREATER => y::ins(x, L)
                      |  _      => x::y::L)

(* isort : int list -> int list
   REQUIRES: true
   ENSURES: isort(L) evaluates to a sorted permutation of L
*)
fun isort [ ] = [ ]
  | isort (x::L) = ins (x, isort L)
```

Let's analyze the work of these two functions. Let $W_{\texttt{ins}}(n)$ denote the work to evaluate `ins(x,L)`, with n the length of `L`. Then, for some constants c_0, c_1, and c_2,

$$\begin{array}{rcll} W_{\texttt{ins}}(0) & = & c_0, & \\ W_{\texttt{ins}}(n) & = & c_1 + W_{\texttt{ins}}(n-1) & \text{when } n > 0 \text{ and } \texttt{x} > \texttt{y}, \\ W_{\texttt{ins}}(n) & = & c_2 & \text{when } n > 0 \text{ and } \texttt{x} \leq \texttt{y}. \end{array}$$

The equation $W_{\texttt{ins}}(n) = c_1 + W_{\texttt{ins}}(n-1)$ describes the worst-case (slowest) scenario when $n > 0$, so we see that $W_{\texttt{ins}}(n)$ is $O(n)$.

Turning to `isort`, let $W_{\texttt{isort}}(n)$ denote the work to evaluate `isort(L)`, with n the length of `L`. Then, for some constants k_0 and k_1,

$$\begin{array}{rcll} W_{\texttt{isort}}(0) & = & k_0, & \\ W_{\texttt{isort}}(n) & = & k_1 + W_{\texttt{isort}}(n-1) + W_{\texttt{ins}}(n-1) & \text{when } n > 0. \end{array}$$

So $W_{\texttt{isort}}(n) \leq W_{\texttt{isort}}(n-1) + k_1 + k_2 * n$, from which we see, by unrolling or by induction, that $W_{\texttt{isort}}(n)$ is $O(n^2)$.

There is no opportunity for parallelism in this code, so the work and span are the same.

We will see more efficient sorting next time.