14
Efficiency, complexity, and recurrences

We say that some agent is *efficient* if it makes the best use of a scarce resource to generate a desired output. Furnaces turn the scarce resource of fuel into heating, so an efficient furnace is one that generates the most heat using the least fuel. Similarly, an efficient shooter in basketball generates the most points using the fewest field goal attempts. Standard measurements of efficiency reflect these notions. Furnaces are rated for Annual Fuel Utilization Efficiency, NBA players for Effective Field Goal Percentage.

Computer programs use scarce resources to generate desired outputs as well. Most prominently, the resources expended are time and "space" (the amount of memory required during the computation), though power is increasingly becoming a resource of interest.

Up to this point, we haven't worried about the efficiency of the programs we've written. And for good reason. Donald Knuth, Professor Emeritus of the Art of Computer Programming at Stanford University and Turing-Award–winning algorithmist, warns of PREMATURE OPTIMIZATION:

> Programmers waste enormous amounts of time thinking about, or worrying about, the speed of noncritical parts of their programs, and these attempts at efficiency actually have a strong negative impact when debugging and maintenance are considered. We *should* forget about small efficiencies, say about 97% of the time: premature optimization is the root of all evil. (Knuth, 1974)

Knuth's point is that *programmers' time is a scarce resource too*, and often the most important one.

Nonetheless, sometimes issues of code efficiency become important – Knuth's 3% – and in any case the special ways of thinking and tools for reasoning about efficiency of computation are important aspects of computational literacy, most centrally ideas of

- Complexity as the *scaling* of resource usage,

Figure 14.1: Donald Knuth (1938–), Professor Emeritus of the Art of Computer Programming at Stanford University. In this photo, he holds a volume of his seminal work *The Art of Computer Programming*.

- Comparison of *asymptotic* scaling,
- *Big-O notation* for specifying asymptotic scaling, and
- *Recurrences* as the means to capture and solve for resource usage.

In this chapter, we describe these important computational notions in the context of an extended example, the comparison of two sorting functions.

14.1 The need for an abstract notion of efficiency

With furnaces and basketball players, we can express a notion of efficiency as a single number – Annual Fuel Utilization Efficiency or Effective Field Goal Percentage. With computer programs, things are not so simple. Consider, for example, one of the most fundamental of all computations, SORTING – ordering the elements of a list according to a comparison function. Given a particular function to sort lists, we can't characterize its efficiency – how long it takes to sort lists – as a single number. What number would we use? That is, how long does it take to sort a list of integers using the function? The answer, of course, is "it depends"; in particular, it depends on

- *Which input?* How many elements are in the list? What order are they in? Are there a lot of duplicate items, or very few?
- *How computed?* Which computer are you using, and which software environment? How long does it take to execute the primitive computations out of which the function is built?

All of these issues affect the running time of a particular sorting function. To make any progress on comparing the efficiency of functions in the face of such intricacy, it is clear that we will need to come up with a more abstract way of characterizing the efficiency of computations.

We address these two issues separately. To handle the question of "which input", we might characterize the efficiency of the sorting program not as a number (a particular running time), but as a *function* from inputs to numbers. However, this doesn't seem an appealing option; we want to be able to draw some general conclusions for comparing sorting programs, not have to reassess for each possible input.

Nonetheless, the idea of characterizing efficiency in terms of some function is a useful one. Broadly speaking, algorithms take longer on bigger problems, so we might use a function that provides the time required as a function of the *size* of the input. In the case of sorting lists, we might take the size of the input to be the number of elements in the list to be sorted. Unfortunately, for any given input size, the program

might require quite different amounts of time. What should we take to be *the* time required for problems of a given size. There are several options: We might consider the time required *on average* for instance. But we will use the time required *in the worst case*. When comparing algorithms, we might well want to plan for the worst case behavior of a program, just to play it safe. We will refer to the function from input sizes to worst-case time needed as the WORST-CASE COMPLEXITY of the algorithm.

We've made some progress. Rather than thinking of resource usage as a single number (too coarse) or a function from problem inputs to numbers (too fine), we use the programs worst-case complexity, a function from sizes of inputs to worst-case resource usage on inputs with those sizes. But even this is not really well defined, because of the "How computed?" question. One and the same program, running on different computers, say, may have wildly different running times.

To make further progress, let's take a concrete example. We'll examine two particular sorting algorithms.

14.2 Two sorting functions

A module signature for sorting can be given by

```
# module type SORT =
#   sig
#     (* sort lt xs -- Returns the list xs sorted in increasing
#        order by the "less than" function lt. *)
#     val sort : ('a -> 'a -> bool) -> 'a list -> 'a list
#   end ;;
module type SORT =
  sig val sort : ('a -> 'a -> bool) -> 'a list -> 'a list end
```

The `sort` function takes as its first argument a comparison function, which specifies when one element should be sorted before another in the desired ordering.

A simple implementation of the signature is the INSERTION SORT algorithm, which operates by inserting the elements of the unsorted list one by one into an empty list, each in its appropriate place.[1]

```
# module InsertSort : SORT =
#   struct
#     let rec insert lt xs x =
#       match xs with
#       | [] -> [x]
#       | hd :: tl -> if lt x hd then x :: xs
#                     else hd :: (insert lt tl x)
#
#     let rec sort (lt : 'a -> 'a -> bool)
#                  (xs : 'a list)
#                : 'a list =
```

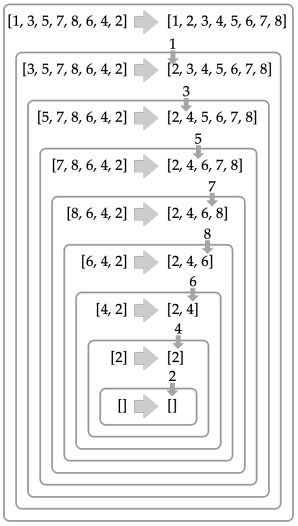

Figure 14.2: An example of the recursive insertion sort algorithm, sorting the list `[1, 3, 5, 7, 8, 6, 4, 2]`. Each recursive call is marked with a rounded box, in which the tail is sorted, and the head then inserted.

[1] Insertion sort could have been implemented more elegantly using a single `fold_left`, but we make the recursion explicit to facilitate the later complexity analysis.

```
#       match xs with
#       | [] -> []
#       | hd :: tl -> insert lt (sort lt tl) hd
#   end ;;
module InsertSort : SORT
```

We can use insertion sort to sort some integers in increasing order:

```
# InsertSort.sort (<) [1; 3; 5; 7; 8; 6; 4; 2] ;;
- : int list = [1; 2; 3; 4; 5; 6; 7; 8]
```

or some floats in decreasing order:

```
# InsertSort.sort (>) [2.71828; 1.41421; 3.14159; 1.61803] ;;
- : float list = [3.14159; 2.71828; 1.61803; 1.41421]
```

An especially elegant implementation of sorting is the MERGE SORT algorithm, first described by John von Neumann in 1945 (according to Knuth (1970)). It works by dividing the list to be sorted into two lists of (roughly) equal size. Each of the halves is then sorted, and the resulting sorted halves are merged together to form the sorted full list. This recursive process of dividing the list in half can't continue indefinitely; at some point the recursion must "bottom out", or the process will never terminate. In the implementation below, we bottom out when the list to be sorted contains at most a single element. The sort function can be defined then as

```
let rec sort lt xs =
  match xs with
  | []
  | [_] -> xs
  | _ -> let first, second = split xs in
         merge lt (sort lt first) (sort lt second) ;;
```

The mergesort definition above makes use of functions

```
split : 'a list -> 'a list * 'a list
```

and

```
merge : ('a -> 'a -> bool) -> 'a list -> 'a list -> 'a
list   .
```

A call to `split lst` returns a pair of lists, each containing half of the elements of `lst`. (In case, `lst` has an odd number of elements, the extra element can go in either list in the returned pair.) A call to `merge lt xs ys` returns a list containing all of the elements of `xs` and `ys` sorted according to `lt`; it assumes that `xs` and `ys` are themselves already sorted.

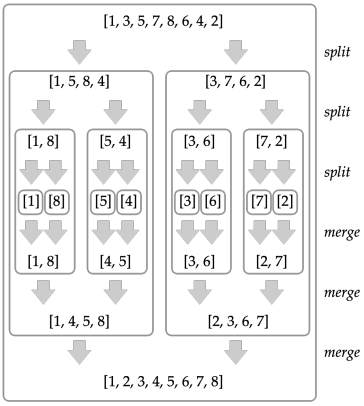

Figure 14.3: An example of the recursive mergesort algorithm, sorting the list `[1, 3, 5, 7, 8, 6, 4, 2]`. Each recursive call is marked with a rounded box, in which the list is split, sorted, and merged.

Exercise 145

Provide implementations of the functions `split` and `merge`, and package them together with the `sort` function just provided in a module `MergeSort` satisfying the `SORT` module type. You should then have a module that allows for the following interactions:

```
# MergeSort.sort (<) [1; 3; 5; 7; 8; 6; 4; 2] ;;
- : int list = [1; 2; 3; 4; 5; 6; 7; 8]
# MergeSort.sort (>) [2.7183; 1.4142; 3.1416; 1.6180] ;;
- : float list = [3.1416; 2.7183; 1.618; 1.4142]
```

(Another elegant recursive sorting algorithm, quicksort, is explored further in Section 16.4.)

14.3 *Empirical efficiency*

How efficient are these algorithms? The time usage of the algorithms can be compared by timing each of them on the same input. Here, we make use of a simple timing function `call_timed : ('a -> 'b) -> 'a -> ('b * float)`. Calling `call_timed f x` evaluates the application of the function `f` to `x`, returning the result paired with the number of milliseconds required to perform the computation.

Now we can sort a list using the two sorting algorithms, reporting the timings as well.[2]

[2] We're taking advantage of several useful functions here. The `map` function from the `List` library module is familiar from Chapter 8. The `Absbook` module, available at http://url.cs51.io/absbookml provides some useful functions that we'll use throughout the book, for example, the `range` function and several timing functions.

```
# (* Generate some lists of random integers *)
# let shortlst = List.init 5 (fun _ -> Random.int 1000) ;;
val shortlst : int list = [344; 685; 182; 641; 439]
# let longlst = List.init 500 (fun _ -> Random.int 1000) ;;
val longlst : int list =
  [500; 104; 20; 921; 370; 217; 885; 949; 678; 615; ...]

# (* test_repeated count f x -- Apply `f` to `x` `count`
#    times, ignoring the results and returning the time
#    taken in milliseconds. *)
# let test_repeated sort lst label =
#   let _, time = Absbook.call_timed
#                   (List.init 1000)
#                   (fun _ -> (sort (<) lst)) in
#   Printf.printf "%-20s %10.4f\n" label time ;;
val test_repeated : (('a -> 'a -> bool) -> 'b -> 'c) -> 'b ->
  string -> unit =
  <fun>

# (* Sort each list two ways *)
# List.iter (fun (sort, lst, label) ->
#            test_repeated sort lst label)
#           [ InsertSort.sort, shortlst, "insertion short";
#             MergeSort.sort, shortlst,  "merge     short";
#             InsertSort.sort, longlst,  "insertion long";
#             MergeSort.sort, longlst,   "merge     long" ] ;;
insertion short          0.6180
merge     short          1.2631
insertion long        3023.9391
merge     long         459.2381
- : unit = ()
```

Not surprisingly, it appears that sometimes the insertion sort algorithm is faster (as on `shortlst`) and sometimes mergesort is faster (as

on `longlst`). It doesn't seem possible to give a definitive answer as to which is faster in general.

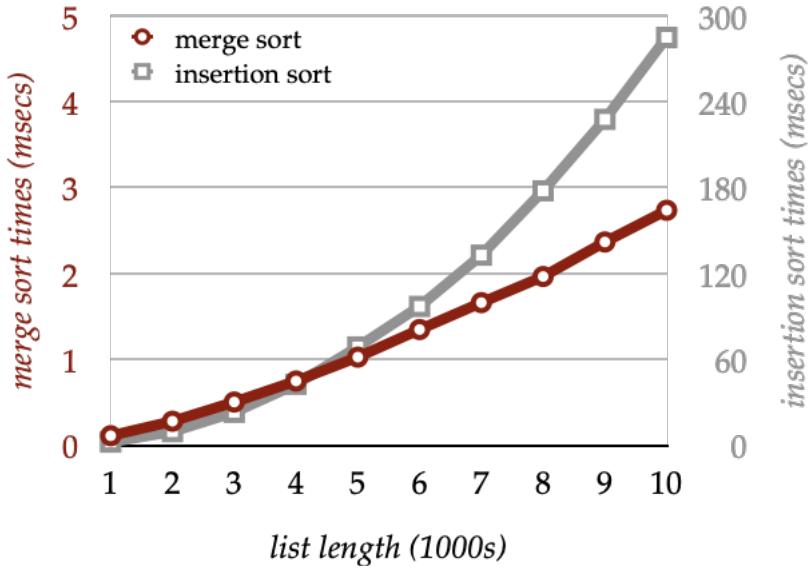

Figure 14.4: Run time in seconds for sorting random lists of lengths varying from 1,000 to 10,000 elements, generated by averaging run time over 100 trials. The two lines show performance for insertion sort and merge sort, with insertion sort times using the right scale to allow for comparison.

If we examine the performance of the algorithm for a broader range of cases, however, a pattern emerges. For short lists, insertion sort is somewhat faster, but as the lists grow in length, the time needed to sort them grows faster for insertion sort than for mergesort, so that eventually mergesort shows a consistent performance advantage. The pattern is quite clear from the graph in Figure 14.4. The key to comparing the algorithms, then, is not their comparative efficiency on any particular list, but rather the character of their efficiency *as their inputs grow in size*. As we argued in Section 14.1, thinking about the time required as a function of the size of the inputs looks like a good idea.

However, as also noted above, a problem with analyzing algorithms, as we have just done, by running them with particular implementations on particular computers on particular lists, is that the results may apply only for those particulars. Instead, we'd like a way of characterizing the algorithms' relative performance whatever the particulars. Measuring running times empirically is subject to idiosyncrasies of the measurement exercise: the relative time required for different primitive operations on the particular computer being used and with the particular software tools, what other operations were happening on the computer at the same time, imprecision in the computer's clock, whether the operating system is slowing down or speeding up the CPU

for energy-saving purposes, and on and on. The particularities also may not be predictive of the future as computers change over time, with processing and memory retrieval and disk accesses becoming faster – and faster at varying rates. The empirical approach doesn't get at the intrinsic properties of the algorithms.

The approach we will take, then, is to analyze the algorithms in terms of the intrinsic growth rate of their performance as the size of their inputs grow, their worst-case complexity. Detailed measurement and analysis can be saved for later, once the more fundamental complexity issues are considered. We thus take an abstract view of performance, rather than a concrete one. This emphasis on abstraction, as usual, comes from thinking like a computer scientist, and not a computer programmer.

The time complexity of the two sorting algorithms can be thought of as functions (!) from the size of the input to the amount of time needed to sort inputs of that size. As it turns out – and as we will show in Sections 14.5.5 and 14.5.9 – for insertion sort on a list of size n, the time required to sort the list grows as the function

$$T_{is}(n) = a \cdot n^2 + b$$

whereas for mergesort, the time required to sort the list grows as the function

$$T_{ms}(n) = c \cdot n \log n + d$$

where a, b, c, and d are some constants. For a given n, which is larger? That depends on these constants of course. But regardless of the constants, as n increases T_{is} grows "faster" than T_{ms} in a way that we will make precise shortly.

In order to make good on this idea of comparing algorithms by comparing their growth functions, then, we must pay on two promissory notes:

1. How to figure out the growth function for a given algorithm, and
2. How to determine which growth functions are growing faster.

In the remainder of this chapter, we will address the first of these with a technique of recurrence equations, and the second with the idea of asymptotic complexity and "big-O" notation.

14.4 Big-O notation

Which is better, an algorithm (like insertion sort) with a complexity that grows as $a \cdot n^2 + b$ or an algorithm (like mergesort) with a complexity that grows as $c \cdot n \log n + d$? The answer "it depends on the values

of the constants" seems unsatisfactory, since intuitively, a function that grows quadratically (as the square of the size) like the former will *eventually* outstrip a function that grows like the latter. Figure 14.5 shows this graphically. The gray lines all grow as $c \cdot n \log n$ for increasing values of c. But regardless of c, the red line, displaying quadratic growth, eventually outpaces all of the gray lines. In a sense, then, we'd *eventually* like to use the $n \log n$ algorithm regardless of the constants. It is this ASYMPTOTIC (that is, long term or eventual) sense that we'd like to be able to characterize.

To address the question of how fast a function grows asymptotically, independent of the annoying constants, we introduce a generic way of expressing the growth rate of a function – BIG-O NOTATION.

We'll assume that problem sizes are non-negative integers and that times are non-negative as well. Given a function f from non-negative integers to non-negative numbers, $O(f)$ is the set of functions that *grow no faster* than f, in the following precise sense:[3] We define $O(f)$ to be the set of all functions g such that for all "large enough" n (that is, n larger than some value n_0), $g(n) \leq c \cdot f(n)$.

The roles of the two constants n_0 and c are exactly to move beyond the details of constants like the a, b, c, and d in the sorting algorithm growth functions. In deciding whether a function grows no faster than f, we don't want to be misled by a few input values here and there where $g(n)$ may happen to be larger than $f(n)$, so we allow exempting values smaller than some fixed value n_0. The point is that as the inputs grow in size, *eventually* we'll get past the few input sizes n where $g(n)$ is larger than $f(n)$. Similarly, if the value of $g(n)$ is always, say, twice the value of $f(n)$, the two aren't growing at qualitatively different rates. Perhaps that factor of 2 is based on just the kinds of idiosyncrasies that can change as computers change. We want to ignore such constant multiplicative factors. For that reason, we don't require that $g(n)$ be less than $f(n)$; instead we require that $g(n)$ be less than *some constant multiple* c of $f(n)$.

As an example of big-O notation, consider two simple polynomial functions. It will be convenient to use Church's elegant lambda notation (see Section B.1.4) to specify these functions directly: $\lambda n.10n^2 + 3$ and $\lambda n.n^2$.

Is the function $\lambda n.10n^2 + 3$ an element of the set $O(\lambda n.n^2)$? To demonstrate that it is, we need to find constants c and n_0 such that for all $n > n_0$, $10n^2 + 3 \leq c \cdot n^2$. It turns out that the values $n_0 = 0$ and $c = 13$ do the trick, that is, for all $n > 0$, $10n^2 + 3 \leq 13n^2$. We can prove this as follows: Since $n \geq 1$, it follows that $n^2 \geq 1$ and thus $3 \leq 3n^2$. Thus $10n^2 + 3 \leq 10n^2 + 3n^2 = 13n^2$. We conclude, then, that

$$\lambda n.10n^2 + 3 \in O(\lambda n.n^2) \quad .$$

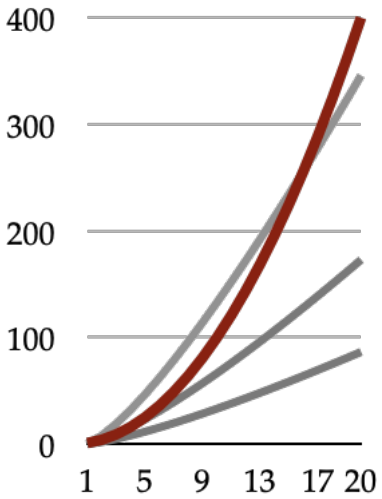

Figure 14.5: A graph of functions with different growth rates. The highlighted line grows as n^2. The three gray lines grow as $c \cdot n \log n$, where c is, from bottom to top, 1, 2, and 4.

[3] Since it takes a function as its argument and returns sets of functions as its output, O is itself a higher-order function!

Of course, the converse is also true:

$$\lambda n.n^2 \in O(\lambda n.10n^2 + 3) \quad .$$

We can just take n_0 again to be 0 and c to be 1, since $n^2 < 10n^2 + 3$ for all n.

14.4.1 Informal function notation

It is conventional, when using big-O notation, to stealthily move between talk of functions (like $\lambda n.n^2$) to the corresponding body expression (like n^2), leaving silent the particular variable (in this case n) that represents the input of the function. Typically, the variable is clear from context (and indeed is frequently the variable n itself). For instance, we might say

$$10n^2 + 3 \in O(n^2) \quad ,$$

rather than the more complex formulation above.

Continuing this abuse of notation, we sometimes write variables for functions, like f or g, not only to stand for a function, but also the corresponding body expression. This allows us to write things like $k \cdot f$ (where k is a constant) to mean the function whose body expression is the product of k and the body expression of f. (This turns out to be equivalent to the rather more cumbersome $\lambda n.k \cdot f(n)$.) Suppose f is the function $\lambda n.n^2$. Then we write $k \cdot f$ to mean, not $k \cdot \lambda n.n^2$ (which is an incoherent formula), but rather $k \cdot n^2$, which, again by convention, glosses $\lambda n.k \cdot n^2$.

This convention of sliding between functions and their body expressions may seem complicated, but it soon becomes quite natural. And it allows us to formulate important properties of big-O notation very simply, as we do in the next section.

Of course, there are problems with playing fast and loose with notation in this way. First, writing $O(n^2)$ makes it look like O is a function that takes integers as input, since n^2 looks like it specifies an integer. But O is a function from *functions*, not from integers. Second, what are we to make of something like $O(m \cdot n^2)$? Does this specify the set of functions that grow no faster than the function from m to $m \cdot n^2$, that is, $O(\lambda m.m \cdot n^2)$? Or does it specify the set of functions that grow no faster than the function from n to $m \cdot n^2$, that is, $O(\lambda n.m \cdot n^2)$? The notation doesn't make clear which variable is the one representing the input to the function – which variable the growth is relative to. In cases such as this, computer scientists rely on context to make clear what the notation is supposed to mean.

We'll stick to this informal notation since it is universally used.

But you'll want to always remember that O maps *functions* to sets of functions.

14.4.2 Useful properties of O

In general, it's tedious to prove particular cases of big-O membership like the example in Section 14.4. Instead, you'll want to acquire a general understanding of these big-O sets of functions, and reason on the basis of that understanding.

The big-O notation brings together whole classes of functions whose growth rates are similar. These classes of functions have certain properties that make them especially useful.[4] First of all, every function grows no faster than itself:

$$f \in O(f)$$

[4] The mathematically inclined might want to take a stab at proving these properties of big-O.

Adding a constant to a function doesn't change its big-O classification: If $g \in O(f)$, then[5]

$$g + k \in O(f) \quad .$$

[5] Here's our first instance of the informal function notation in the wild.

We can reason immediately, then, that $2n^2 + 3 \in O(2n^2)$ (or, more pedantically, $\lambda n.2n^2 + 3 \in O(\lambda n.2n^2)$), without going through a specific proof.

Similarly, multiplying by a constant (k) doesn't affect the class either. If $g \in O(f)$, then

$$k \cdot g \in O(f) \quad .$$

Thus, $2n^2 \in O(n^2)$. Together with the results above, we can conclude that $2n^2 + 3 \in O(n^2)$.

In fact, adding in any lower degree terms doesn't matter. If $f \in O(n^k)$ and $g \in O(n^c)$, where $k > c$:

$$f + g \in O(n^k)$$

The upshot of all this is that in determining the big-O growth rate of a polynomial function, we can always just drop lower degree terms and multiplicative constants. In thinking about the growth rate of a complicated function like $4n^3 + 142n + 3$, we can simply ignore all but the largest degree term ($4n^3$) and even the multiplicative constant 4, and conclude that

$$4n^3 + 142n + 3 \in O(n^3)$$

Exercise 146

Which of these claims about the growth rates of various functions hold?

1. $3n+5 \in O(n)$
2. $n \in O(3n+5)$
3. $n+n^2 \in O(n)$
4. $n^3+n^2 \in O(n^3+2n)$
5. $n^2 \in O(n^3)$
6. $n^3 \in O(n^2)$
7. $32n^3 \in O(n^2+n+k)$

Finally, the sum or product of functions grows no faster than the sum or product, respectively, of their respective growth rates. If $f' \in O(f)$ and $g' \in O(g)$, then

$$f'+g' \in O(f+g)$$
$$f' \cdot g' \in O(f \cdot g)$$

We can thus conclude that

$$(5n^3+n^2) \cdot (3\log n+7) \in O(n^3 \cdot \log n)$$

14.4.3 Big-O as the metric of relative growth

We are interested in the big-O classification of functions in particular because we can use it to compare functions as to which asymptotically grows faster. In particular, if $f \in O(g)$ but $g \notin O(f)$, then g grows faster than f, which we notate $g \gg f$.

For example,

$$n^2 \in O(n^3) \quad ,$$

but the converse doesn't hold:

$$n^3 \notin O(n^2) \quad .$$

We can conclude, then that

$$n^3 \gg n^2 \quad ,$$

that is, n^3 grows faster than n^2.

More generally,

- Functions with bigger exponents grow faster:

  $$n^k \gg n^c \quad \text{when } k > c$$

- Linear functions grow faster than logarithmic functions:

  $$n \gg \log n$$

- Exponentials grow faster than polynomials:

  $2^n \gg n^k$

- The exponential base matters; exponentials with larger base grow faster:

  $3^n \gg 2^n$

We can think of big-O as defining classes of functions that grow at similar rates, up to multiplicative constants. Thus, $O(n)$ is the set of functions whose growth rate is (at most) linear, $O(n^2)$ the set of functions whose growth rate is (at most) quadratic, $O(n^3)$ the set of cubic functions, $O(2^n)$ the set of base-two exponential functions. We can then place these classes in an ordering ($\gg$) as to which classes grow faster inherently (and not just because of the values of some contingent constants).

From the properties above, we can conclude that $n^2 \gg n\log n$, and therefore, since $T_{is} \in O(n^2)$ and $T_{ms} \in O(n\log n)$, that $T_{is} \gg T_{ms}$.[6] Mergesort has lower complexity – is asymptotically more efficient – than insertion sort. This conclusion is independent of which computers we time the algorithms on, or other particularities that affect the constants.

[6] Strictly speaking, we'd have to further show that $T_{is} \notin O(n\log n)$, but we'll ignore this nicety in general here and in the following discussion.

Of course, this reasoning relies on knowing the functions for how each algorithm's performance scales. (In the discussion above, we merely asserted the growth rate functions for the two sorting algorithms.) Only then can we use big-O to determine which algorithm scales better, which is more efficient in a deeper sense than just testing a particular instance or two. We still need a way to determine for a particular algorithm the particular resource-usage function. This is the second promissory note, and the one that we now address.

Problem 147

Two friends who work at EuclidCo tell you that they're looking for a fast algorithm to solve a problem they're working on. So far, they've each developed an algorithm: algorithm A has time complexity $O(n^3)$ and algorithm B is $O(2^n)$. They prefer algorithm A, and use three different arguments to convince you of their preference. For each argument, evaluate the truth of the bolded statement, and justify your answer.

1. "We're all about speed at EuclidCo, and **A will always be faster than B.**"
2. "In a high stakes industry like ours, we can't afford to have more than a finite number of inputs that run slower than polynomial time, and **we can avoid this if we go with A.**"
3. "We work with big data at EuclidCo. **For suitably large inputs, A will be faster on average than B.**"

14.5 Recurrence equations

Given an algorithm, how are we to determine how much time it needs as a function of the size of its input? In this section, we introduce one

method, based on the solving of recurrence equations, to address this question.

We start with a simple example, the `append` function to append two lists, defined as

```
# let rec append xs ys =
#   match xs with
#   | [] -> ys
#   | hd :: tl -> hd :: (append tl ys) ;;
val append : 'a list -> 'a list -> 'a list = <fun>
```

An appropriate measure for the size of the input to the function is the sizes of the two lists it is to append. Let's use $T_{append}(n, m)$ for the time required to run the `append` function on lists with n and m elements respectively. What do we know about this T_{append}?

When the first argument, `xs`, is the empty list (so $n = 0$), the function performs just a few simple actions, pattern-matching the input against the empty list pattern, and then returning `ys`. If we say that the time for the pattern match is some constant c_{match} and the time for the return is some constant $c_{returnys}$, then we have that

$$T_{append}(0, m) = c_{match} + c_{returnys} \quad .$$

Since the sum of the two constants is itself a constant, we can simplify by treating the whole as a new constant c:

$$T_{append}(0, m) = c$$

When the first argument is nonempty, the computation performed again has a few parts: the match against the first pattern (which fails), the match against the second pattern (which succeeds), the recursive call to `append`, the cons of `h` and the result of the recursive call. Each of these (except for the recursive call) takes some constant time, so we can characterize the amount of time as

$$T_{append}(n + 1, m) = c_{matchcons} + T_{append}(n, m) \quad .$$

Here, we use $n + 1$ as the length of the first list, as we know it is at least one element long. The recursive call is appending the tail of `xs`, a list of length n, to `ys`, a list of length m, and thus (by hypothesis) takes time $T_{append}(n, m)$.

Merging and renaming constants, we thus have the following two RECURRENCE EQUATIONS that characterize the running time of the `append` function in terms of the size of its arguments:

$$\begin{aligned} T_{append}(0, m) &= c \\ T_{append}(n + 1, m) &= k + T_{append}(n, m) \end{aligned}$$

14.5.1 Solving recurrences by unfolding

Because the recurrence equations defining T_{append} use T_{append} itself, recursively, in the definition (hence the term "recurrence"), they don't provide a CLOSED-FORM (nonrecursive) solution to the question of characterizing the running time of the function. To get a solution in closed form, we will use a method called UNFOLDING to solve the recurrence equations.

Consider the general case of $T_{append}(n, m)$ and assume that $n > 0$. By the second recurrence equation,

$$T_{append}(n, m) = k + T_{append}(n-1, m) \quad .$$

Now $T_{append}(n-1, m)$ itself can be unfolded as per the second recurrence equation, so

$$T_{append}(n, m) = k + k + T_{append}(n-2, m) \quad .$$

Continuing in this vein, we can continue to unfold until the first argument to T_{append} becomes 0:

$$\begin{aligned} T_{append}(n, m) &= k + T_{append}(n-1, m) \\ &= k + k + T_{append}(n-2, m) \\ &= k + k + k + T_{append}(n-3, m) \\ &= \cdots \\ &= k + k + k + \cdots + k + T_{append}(0, m) \end{aligned}$$

How many unfoldings are required until the first argument reaches 0? We'll have had to unfold n times. There will therefore be n instances of k being summed in the unfolded equation. Completing the derivation, then, using the first recurrence equation,

$$\begin{aligned} T_{append}(n, m) &= k \cdot n + T_{append}(0, m) \\ &= k \cdot n + c \end{aligned}$$

We now have the closed-form solution

$$T_{append}(n, m) = k \cdot n + c$$

Notice that the time required is independent of m, the size of the second argument. That makes sense because the code for `append` never looks inside the structure of the second argument; the computation therefore doesn't depend on its size.

Now, the function $k \cdot n + c \in O(n)$. Thus the time complexity of `append` is $O(n)$ or *linear* in the length of its first argument. This is typical of algorithms that operate by recursively marching down a list one element at a time.

In order to apply the same kinds of techniques to determine the time complexity of the two sorting algorithms, we'll work through a series of examples.

14.5.2 Complexity of reversing a list

There are multiple ways of implementing list reversal. We show that they can have quite different time complexities. We start with a naive implementation, which works by reversing the tail of the list and appending the head on the end:

```
# let rec rev xs =
#   match xs with
#   | [] -> []
#   | hd :: tl -> append (rev tl) [hd] ;;
val rev : 'a list -> 'a list = <fun>
```

We define recurrence equations for the time $T_{rev}(n)$ to reverse a list of length n using this implementation. If the list is empty, we have (similarly to the case of `append`, and introducing constants as needed):

$$T_{rev}(0) = c_{match} + c_{return} = q$$

For nonempty lists, we must perform the appropriate match, reverse the tail, cons the head onto the empty list, and perform the append:

$$\begin{aligned} T_{rev}(n+1) &= c_{match} + c_{cons} + T_{rev}(n) + T_{append}(n,1) \\ &= r + T_{rev}(n) + T_{append}(n,1) \\ &= r + T_{rev}(n) + k \cdot n + c \\ &= k \cdot n + s + T_{rev}(n) \end{aligned}$$

The closed form solution for `append` from the previous section becomes useful here. And again, notice our free introduction of new constants to simplify things. We take the sum of c_{match} and c_{cons} to be r, then for $r + c$ we introduce s. Summarizing, the `reverse` implementation above yields the recurrence equations

$$\begin{aligned} T_{rev}(0) &= q \\ T_{rev}(n+1) &= k \cdot n + s + T_{rev}(n) \end{aligned}$$

which we must now solve to find a closed form.

We again unfold $T_{rev}(n)$:

$$\begin{aligned}
T_{rev}(n) &= k \cdot (n-1) + s + T_{rev}(n-1) \\
&= k \cdot (n-1) + s + k \cdot (n-2) + s + T_{rev}(n-2) \\
&= k \cdot (n-1) + s + k \cdot (n-2) + s \\
&\qquad + k \cdot (n-3) + s + T_{rev}(n-3) \\
&= \cdots \\
&= k \cdot \sum_{i=1}^{n-1} (n-i) + s \cdot n + T_{rev}(0) \\
&= k \cdot \sum_{i=1}^{n-1} (n-i) + s \cdot n + q \\
&= k \cdot \sum_{i=1}^{n-1} i + s \cdot n + q \\
&= k \cdot \sum_{i=1}^{n} i - k \cdot n + s \cdot n + q \\
&= k \cdot \sum_{i=1}^{n} i + (s-k) \cdot n + q
\end{aligned}$$

To make further progres on achieving a simple closed form for the recurrence, it would be ideal to simplify the summation $\sum_{i=1}^{n} i$ of the integers from 1 to n. Famously (if apocryphally), the seven-year-old mathematical prodigy Carl Friedrich Gauss (1777–1855) solved this problem in his head. Gauss was asked by his teacher, so the story goes, to sum all of the integers from 1 to 100. That'll keep him quiet for a bit, the teacher presumably thought. But Gauss came up with the answer – 5050 – immediately, by taking advantage of the simple identity

$$\sum_{i=1}^{n} i = \frac{n \cdot (n+1)}{2}$$

For a graphical "proof" that the identity holds, see Figure 14.6. A more traditional proof is provided in Section B.2.

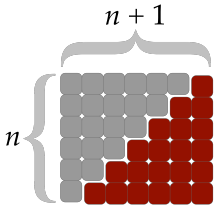

Figure 14.6: A graphical proof that $\sum_{i=1}^{n} i = \frac{n \cdot (n+1)}{2}$.

Two triangles, each formed by piling up squares with rows from 1 to n can be combined to form a rectangle of area $n \cdot (n+1)$. Each triangle is half that area, that is, $\frac{n \cdot (n+1)}{2}$. A more algebraic proof is given in Section B.2.

Making use of this identity,

$$\begin{aligned}
T_{rev}(n) &= k \cdot \sum_{i=1}^{n} i + (s-k) \cdot n + q \\
&= k \cdot \frac{n \cdot (n+1)}{2} + (s-k) \cdot n + q \\
&= \frac{k}{2} n^2 + \frac{k}{2} n + (s-k) \cdot n + q \\
&= \frac{k}{2} n^2 + (s - \frac{k}{2}) \cdot n + q \\
&\in O(n^2)
\end{aligned}$$

concluding that the `function` has quadratic ($O(n^2)$) complexity. The last step really shows the power of big-O notation, allowing to strip

away all of the constants and lower order terms to get at the essence of the growth rate.

Problem 148

Recall that the `Stdlib.compare` function compares two values, returning an `int` based on their relative magnitude: `compare x y` returns `0` if `x` is equal to `y`, `-1` if `x` is less than `y`, and `+1` if `x` is greater than `y`.

A function `compare_lengths : 'a list -> 'b list -> int` that compares the lengths of two lists can be implemented using `compare` by taking advantage of the `length` function[7] from the `List` module:

```
let compare_lengths xs ys =
  compare (List.length xs) (List.length ys) ;;
```

For instance,

```
# compare_lengths [1] [2; 3; 4] ;;
- : int = -1
# compare_lengths [1; 2; 3] [4] ;;
- : int = 1
# compare_lengths [1; 2] [3; 4] ;;
- : int = 0
```

However, this implementation of `compare_lengths` does a little extra work than it needs to. Its complexity is $O(n)$ where n is the length of the *longer* of the two lists.

Why does `compare_lengths` have this big-O complexity? In particular, why does the length of the shorter list not play a part in the complexity? We're looking for a brief informal argument here, not a full derivation of its complexity.

Provide an alternative implementation of `compare_lengths` whose complexity is $O(n)$ where n is the length of the *shorter* of the two lists, not the `longer`.

[7] For reference, this built-in `length` function is, unsurprisingly, linear in the length of its argument.

14.5.3 Complexity of reversing a list with accumulator

An alternative method of reversing a list uses an accumulator. As each element in the list is processed, it is consed on the front of the accumulating list. The process begins with the empty accumulator.

```
# let rec revappend xs accum =
#   match xs with
#     | [] -> accum
#     | hd :: tl -> revappend tl (hd :: accum) ;;
val revappend : 'a list -> 'a list -> 'a list = <fun>

# let rev xs = revappend xs [] ;;
val rev : 'a list -> 'a list = <fun>
```

As before, we can set up recurrence equations for this version of `rev` and its auxiliary function `revappend`.

$$T_{rev}(n) = q + T_{revapp}(n, 0)$$

$$T_{revapp}(0, m) = c$$

$$T_{revapp}(n+1, m) = k + T_{revapp}(n, m+1)$$

By an unfolding argument similar to that for `append`, we can solve these recurrence equations to closed form:

$$\begin{aligned} T_{revapp}(n,m) &= k \cdot n + c \\ &\in O(n) \end{aligned}$$

so that

$$\begin{aligned} T_{rev}(n) &= q + T_{revapp}(n,0) \\ &= q + k \cdot n + c \\ &\in O(n) \end{aligned}$$

Unlike the quadratic simple `reverse`, the `revappend` approach is linear. The difference is born out empirically as well, as shown in Figure 14.7.

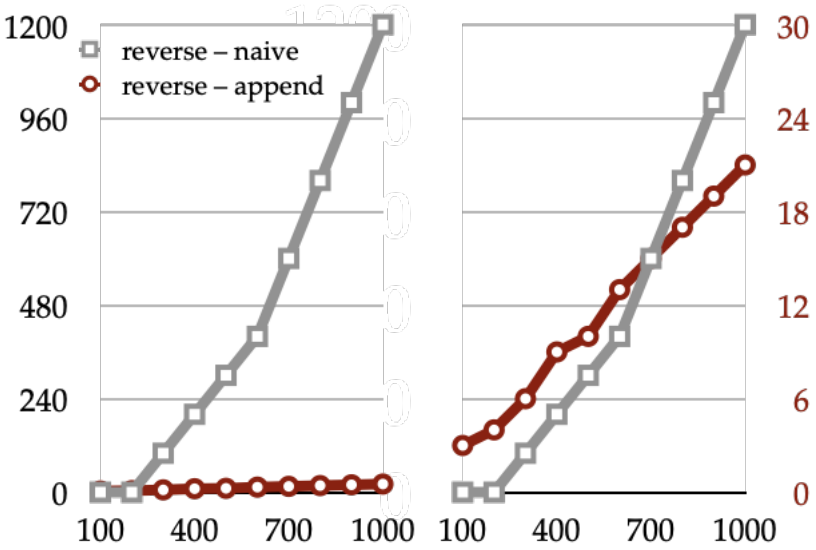

Figure 14.7: Time in microseconds to reverse lists of lengths 100 to 1000 using the naive (square) and revappend (circle, highlighted) implementations. The left graph places both lines on the same (left) vertical scale. The right graph places the revappend line on the right vertical scale (equivalent to multiplying all of the revappend times by 50) to emphasize the difference in growth rate of the functions. Despite the change in constants, the naive reverse still eventually overtakes the revappend.

14.5.4 Complexity of inserting in a sorted list

The insertion sort algorithm uses a function `insert` to insert an element in its place in a sorted list:

```
# let rec insert xs x =
#   match xs with
#   | [] -> [x]
#   | hd :: tl -> if x > hd then hd :: (insert tl x)
#                 else x :: xs ;;
val insert : 'a list -> 'a -> 'a list = <fun>
```

As usual, we construct appropriate recurrence equations for $T_{insert}(n)$ where n is the length of the list being inserted into. (We ignore the element argument, as its size is irrelevant to the time required.) Inserting into the empty list takes constant time.

$$T_{insert}(0) = c$$

Inserting into a nonempty list (of size $n+1$) is more subtle. The time required depends on whether the element should come at the start of the list (the *else* clause of the conditional) or not (the *then* clause). In the former case, the cons operation takes constant time, say k_2; in the latter case, it involves a recursive call to `insert` ($T_{insert}(n)$) plus some further constant overhead (k_1). Since we don't know which way the computation will branch, we have to make the worst-case assumption: whichever is bigger. Which of the two is bigger depends on the constants, but we can be sure, in any case, that the time required is certainly less than the sum of the two.

$$\begin{aligned} T_{insert}(n+1) &= \max(k_1 + T_{insert}(n), k_2) \\ &\leq k_1 + T_{insert}(n) + k_2 \\ &= k + T_{insert}(n) \end{aligned}$$

Unfolding these proceeds as usual:

$$\begin{aligned} T_{insert}(n) &= k + T_{insert}(n-1) \\ &= k + k + T_{insert}(n-2) \\ &= \cdots \\ &= k \cdot n + T_{insert}(0) \\ &= k \cdot n + c \\ &\in O(n) \end{aligned}$$

Insertion is thus linear in the size of the list to be inserted into.

14.5.5 Complexity of insertion sort

Recall the implementation of insertion sort:

```
let rec sort (lt : 'a -> 'a -> bool)
             (xs : 'a list)
           : 'a list =
  match xs with
  | [] -> []
  | hd :: tl -> insert lt (sort lt tl) hd ;;
```

Using similar arguments as above, the recurrence equations can be determined to be

$$\begin{aligned} T_{isort}(0) &= c \\ T_{isort}(n+1) &= k + T_{isort}(n) + T_{insert}(n) \end{aligned}$$

Solving the recurrence equations:

$$\begin{aligned} T_{isort}(n) &= k + T_{isort}(n-1) + O(n-1) \\ &= k + k + T_{isort}(n-2) + O(n-1) + O(n-2) \\ &= k \cdot n + T_{isort}(0) + O(n-1) + O(n-2) + \cdots + O(0) \\ &= k \cdot n + c + \sum_{i=1}^{n} O(i) \\ &\in O(n^2) \end{aligned}$$

We conclude that insertion sort is quadratic in its run time.

14.5.6 Complexity of merging lists

Continuing our exploration of the time complexity of sorting algorithms, we turn to the components of mergesort. The `merge` function, defined by

```
let rec merge lt xs ys =
  match xs, ys with
  | [], _ -> ys
  | _, [] -> xs
  | x :: xst, y :: yst ->
     if lt x y
     then x :: (merge lt xst ys)
     else y :: (merge lt xs yst) ;;
```

takes two list arguments; their sizes will be two of the arguments of the complexity function T_{merge}. Each recursive call of `merge` reduces the total number of items in the two lists. We will for that reason use the sum of the sizes of the two lists as the argument to T_{merge}.

If the total number of elements in the two lists is 1, then one of the two lists must be empty, and we have

$$T_{merge}(1) = c$$

In the worst case, neither element will become empty until the total number of elements in the lists is 2. Thus, for $n \geq 2$, we have the "normal" case, when the lists are nonempty, which involves (in addition to some constant overhead) a recursive call to `merge` with one fewer element in the lists. In the worst case, both elements will still be nonempty.

$$T_{merge}(n+1) = k + T_{merge}(n)$$

Solving these recurrence equations:

$$\begin{aligned} T_{merge}(n) &= k + T_{merge}(n-1) \\ &= k + k + T_{merge}(n-2) \\ &= \cdots \\ &= k \cdot n + T_{merge}(1) \\ &= k \cdot n + c \\ &\in O(n) \end{aligned}$$

14.5.7 Complexity of splitting lists

We leave as an exercise to show that the `split` function defined by

```
let rec split lst =
  match lst with
  | []
  | [_] -> lst, []
  | first :: second :: rest ->
      let first', second' = split rest in
      first :: first', second :: second' ;;
```

has linear time complexity.

Exercise 149

Show that `split` has time complexity linear in the size of its first list argument.

14.5.8 Complexity of divide and conquer algorithms

Before continuing to the analysis of mergesort, we look more generally at algorithms that (like mergesort) attack problems by dividing them into equal parts, recursively solving them, and putting the subsolutions back together to solve the full problem – DIVIDE-AND-CONQUER algorithms.

The recurrences of such algorithms are typically structured with a base case requiring constant time

$$T(1) = c$$

and a recursive case that involves two recursive calls on some problems each of half the size. At first, we'll assume that the time to break apart and put together the two parts takes constant time k.

$$T(n) = k + 2 \cdot T(n/2)$$

For simplicity in solving these recurrence equations, we assume that

n is a power of 2. Then unfolding a few times:

$$\begin{aligned} T(n) &= k + 2 \cdot T(n/2) \\ &= k + k + 4 \cdot T(n/4) \\ &= k + k + k + 8 \cdot T(n/8) \\ &= \cdots \end{aligned}$$

How many times can we unfold? The denominator keeps doubling. We can keep doubling, then, m times until $2^m = n$, that is, $m = \log n$:

$$\begin{aligned} T(n) &= k + 2 \cdot T(n/2) \\ &= k + k + 4 \cdot T(n/4) \\ &= k + k + k + 8 \cdot T(n/8) \\ &= \cdots \\ &= k \cdot \log n + n \cdot T(n/n) \\ &= k \cdot \log n + c \cdot n \\ &\in O(n) \end{aligned}$$

($\log n$ times)

More realistically, however, the time required to divide the problem up and to merge the subsolutions together may take time linear in the size of the problem. In that case, the recurrence would be something like

$$T(n) = k \cdot n + 2 \cdot T(n/2)$$

and the closed form is derived as

$$\begin{aligned} T(n) &= k \cdot n + 2 \cdot T(n/2) \\ &= k \cdot n + k \cdot n + 4 \cdot T(n/4) \\ &= k \cdot n + k \cdot n + k \cdot n + 8 \cdot T(n/8) \\ &= \cdots \\ &= k \cdot n \cdot \log n + n \cdot T(n/n) \\ &= k \cdot n \cdot \log n + c \cdot n \\ &\in O(n \log n) \end{aligned}$$

($\log n$ times)

The $O(n \log n)$ complexity is the hallmark of divide-and-conquer algorithms. Since $\log n$ grows extremely slowly, such algorithms are almost linear in their complexity, thus very efficient.

14.5.9 Complexity of mergesort

Having determined the time complexity for the components of merge-sort, we put them together to determine the complexity of the merge-sort function itself:

```
let rec msort xs =
  match xs with
  | [] -> xs
  | [_] -> xs
  | _ -> let fst, snd = split xs in
         merge (msort fst) (msort snd) ;;
```

$$T_{msort}(0) = T_{msort}(1) = c_1$$
$$T_{msort}(n) = c_2 + T_{split}(n) + 2 \cdot T_{msort}(n/2) + T_{merge}(n)$$

Since both T_{split} and T_{merge} are linear, we can write

$$T_{msort}(n) = k \cdot n + c + 2 \cdot T_{msort}(n/2)$$

These recurrence equations are just of the divide-and-conquer sort, so we know immediately that the complexity of mergesort is $O(n \log n)$. And since

$$n^2 \gg n \log n$$

mergesort is shown to be asymptotically more efficient than insertion sort.

Consistent with this analysis of the sorting algorithms is their empirical performance, as shown in Figure 14.4. The figure depicts well the almost linear behavior of mergesort and the much steeper quadratic growth of insertion sort.

14.5.10 Basic Recurrence patterns

Table 14.1 summarizes some of the basic types of recurrence equations and their closed-form solution in terms of big-*O*.

Table 14.1: Some common recurrence patterns and their closed-form solution in terms of big-*O*.

$T(n) = c + T(n-1)$	$T(n) \in O(n)$
$T(n) = c + k \cdot n + T(n-1)$	$T(n) \in O(n^2)$
$T(n) = c + k \cdot n^d + T(n-1)$	$T(n) \in O(n^{d+1})$
$T(n) = c + 2 \cdot T(n/2)$	$T(n) \in O(n)$
$T(n) = c + T(n/2)$	$T(n) \in O(\log n)$
$T(n) = c + k \cdot n + 2 \cdot T(n/2)$	$T(n) \in O(n \cdot \log n)$

14.6 Problem section: Complexity of the Luhn check

Recall the Luhn check algorithm from Section 8.5, and its various component functions: `evens`, `odds`, `doublemod9`, `sum`.

Problem 150

What is an appropriate recurrence equation for defining the time complexity of the `odds` function from Problem 60 in terms of the length of its list argument?

Problem 151

What is the time complexity of the `odds` function from Problem 60 (in big-O notation)?

Problem 152

If the function $f(n)$ is the time complexity of `odds` on a list of n elements, which of the following is true?

- $f \in O(1)$
- $f \in O(\log n)$
- $f \in O(\log n / c)$ for all $c > 0$
- $f \in O(c \cdot \log n)$ for all $c > 0$
- $f \in O(n)$
- $f \in O(n / c)$ for all $c > 0$
- $f \in O(c \cdot n)$ for all $c > 0$
- $f \in O(n^2)$
- $f \in O(n^2 / c)$ for all $c > 0$
- $f \in O(c \cdot n^2)$ for all $c > 0$
- $f \in O(2^n)$
- $f \in O(2^n / c)$ for all $c > 0$
- $f \in O(c \cdot 2^n)$ for all $c > 0$

Problem 153

What is the time complexity of the `luhn` function implemented in Problem 63 in terms of the length n of its list argument? Use big-O notation. Explain why your implementation has that complexity.

14.7 Supplementary material

- Lab 10: Time complexity, big-O, and recurrence equations
- Problem set A.6: The search for intelligent solutions