15-150 Fall 2013
Lecture 5

Stephen Brookes

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 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 introducing an "accumulator", or by computing extra information. We give some examples.
- Mathematical insight may also lead to more efficient code.

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 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.

big-O classification

Asymptotic analysis is based on big-O classifications: $\texttt{O}(1)$ or "constant time"; $\texttt{O}(n)$ or "linear"; $\texttt{O}(n^2)$ or "quadratic"; $\texttt{O}(log\ n)$, 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 $\texttt{O}(n^3)$ 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 $\texttt{O}(n^3)$ is better than or faster than $\texttt{O}(2^n)$.

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

And 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 $\texttt{O}(n^2)$, 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". At the end of the notes for today we summarize some key results.

4 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 resist the temptation to include REQUIRES and ENSURES comments, again because the main focus here is on runtime, not on correctness. Nevertheless we try to give clear informal specifications, and we indicate how you could prove correctness.

In many of these examples, we 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.) But we are concerned here with analyzing *how long* it takes for function calls to compute their results.

4.1 Powers of 2

Here is a simple ML function `exp` for calculating powers of 2.

```
(* exp : int -> int *)
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 $n \geq 0$, `exp` $n = 2^n$.

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. It should then be clear from the structure of the function definition that there are (non-negative) constants c_0, c_1 such that

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

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

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

It is also easy to show, using this closed form, that $W_{\texttt{exp}}(n)$ is $\texttt{O}(n)$. Here is a sketch of the details. 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 $N = c_0$. 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 $\texttt{O}(n)$. In other words, the running time for `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 $\texttt{c}_0$ and $\texttt{c}_1$, 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

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

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 (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 $n \geq 0$, `fastexp` $n = 2^n$.

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:

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

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:

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

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

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

$T_{\texttt{fastexp}}$ defined this way is obviously not the same function as $W_{\texttt{fastexp}}$ 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 $T_{\texttt{fastexp}}$.

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

This doesn't imply that $W_{\texttt{fastexp}}(n)$ 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. That means that $W_{\texttt{fastexp}}(n)$ is in the same `O`-class as $T_{\texttt{fastexp}}(n)$. Hence $W_{\texttt{fastexp}}(n)$ is $\texttt{O}(log_2\ n)$.

Recall another well known property of big-O notation: $\texttt{O}(log_2\ n)$ means the same as $\texttt{O}(log_3\ n)$, and so on. 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, faster or slower

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

```
(* pow : int -> int *)
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;
```

Notice that we use a local variable `k` to hold the value returned by the recursive call, and this variable gets used twice (in each branch). This fact turns out to be crucial in our analysis!

Again it is easy to prove by induction on n that for all $n \geq 0$, `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 $W_{\texttt{pow}}(n)$ be the runtime of `pow` n, for $n \geq 0$. The function definition tells us that there are constants c_0, c_1, c_2 such that:

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

This is essentially the same recurrence as the one for $W_{\texttt{fastexp}}$, so the runtime of `pow` n is $\texttt{O}(log\ n)$, the same as for `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. Here is a bad version that makes redundant recursive calls. Compare the code with that of `pow`.

```
(* badexp : int -> int *)
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 $W_{\texttt{badpow}}(n)$ be the runtime of `badpow` n, for $n \geq 0$. Then (again, from the function definition) we can derive the recurrence

$$\begin{aligned} W_{\texttt{badpow}}(0) &= 1 \\ W_{\texttt{badpow}}(1) &= 1 \\ W_{\texttt{badpow}}(n) &= 1 + 2 * W_{\texttt{badpow}}(n \texttt{ div } 2) \text{ if } n > 1 \end{aligned}$$

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

$$\begin{aligned} W_{\texttt{badpow}}(2^0) &= 1 \\ W_{\texttt{badpow}}(2^1) &= 1 + 2W_{\texttt{badpow}}(2^0) = 1 + 2 = 3 \\ W_{\texttt{badpow}}(2^2) &= 1 + 2W_{\texttt{badpow}}(2^1) = 1 + 2 + 4 = 7 \end{aligned}$$

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

4.4 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, $b^n = (b^2)^{n \texttt{ div } 2}$. We were unable to take advantage of this particular kind of math fact earlier, because we were fixated on computing powers of 2. By tackling a more general task we actually make it possible to exploit results from math to develop a faster algorithm.

```
(* gexp : int * int -> int *)
fun gexp (b, 0) = 1
 |  gexp (b, 1) = b
 |  gexp (b, n) =
      let
         val k = gexp (b*b, n div 2)
      in
          if n mod 2 = 0 then k else b*k
      end;
```

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

4.5 Fibonacci numbers

Here is a simple ML definition 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 ML interpreter window we will see that `fib` 42 takes a very long time to return its result; and `fib` 43 raises the `Overflow` exception, because the 43^{rd} 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

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

While this recurrence doesn't seem easily solvable (at least, not explicitly), it is obvious that `fib`$(n) \leq W_{\texttt{fib}}(n)$ for all $n \geq 0$. And mathematicians have proven that `fib`(n) is exponential in n, so $W_{\texttt{fib}}$ 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` indeed has exponential running time.

We can speed up the code by computing two Fibonacci numbers in each iteration; we introduce a helper function that does this.

```
(* fastfib : int -> int *)
(* Local function loop : int * int * int -> int *)
(* Counts from n down to 0 *)
fun fastfib n =
  let
    fun loop(i, a, b) = if i=0 then a else loop(i-1, b, a+b)
  in
    loop(n, 1, 1)
  end;
```

Let $W_{\texttt{loop}}(n)$ be the running time for $\texttt{loop}(n, a, b)$, 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

$$\begin{aligned} W_{\texttt{loop}}(0) &= c_0 \\ W_{\texttt{loop}}(i) &= c_1 + W_{\texttt{loop}}(i-1) \text{ for } i > 0 \end{aligned}$$

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

Actually, `fastfib 42` returns the result very quickly, but `fastfib 43` raises the `Overflow` exception because the 43rd Fibonacci number is too large.

Note that the functions `fib` and `fastfib` are extensionally equivalent, even though their runtimes differ significantly.

Exercise: prove that `fib` = `fastfib`. You will need to state and prove a suitable specification for `loop`.

4.6 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 expression `x::E` takes time proportional to the sum of the times to evaluate `x` and `E`, plus a constant for the final `::` operation.[1] 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.

```
(* rev : int list -> int list *)
(* REQUIRES true *)
(* ENSURES rev(L) = the reverse list of L *)
fun rev [ ] = [ ]
 |  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 `rev [1,2,3] = [3,2,1]`.

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

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

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

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

[1] 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.

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 $n \geq 0$,

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

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

Faster reversal

Sometimes there is a fairly obvious way to improve efficiency by introducing an extra argument to the function, and using it to "accumulate" or build up the result.

A faster way to append two lists: use an accumulator list to build up the result. We can build up the result list with `::` instead of @.

```
(* revver : int list * int list -> int list *)
fun revver([ ], A) = A
 |  revver(x::L, A) = revver(L, x::A);

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

It is easy to show (by list induction on `L`) that for all integer lists L, A, `revver`$(L, A) =$ (`rev` L)@A. Hence, `Rev` $L =$ `rev` L, for all L.

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

The runtime of `Rev(L)` is linear in the length of `L`.

Warning

Introducing an accumulator to improve efficiency is a widely useful technique. But this is not a panacea: the trick doesn't always truly improve runtime! Consider:

```
(* exp' : int * int -> int *)
fun exp' (n, a) = if n=0 then a else exp' (n-1, 2*a);
fun Exp n = exp'(n, 1);
```

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

Note: as we will see later, depending on the setting, you may need to choose the accumulator to be a list, an integer, a function, or a value of some other type.

5 Work and span

So far we have talked about running time for code executed 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 even 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: obviously a step that depends on or uses data from another step in the computation must occur later, when the data is available. For example, if we add a collection of integer rows, followed by adding the row sums, the second phase depends on the rests of the row calculations, so the span of this method of summing is proportional to $\sum_{i=1}^{k} n_k + k$, where k is the number of rows and n_i is the length of the i^{th} row. (We assumed here that the row calculations are each done sequentially, so that the time to sum a list of n_i items is proportional to n_i.)

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

Later we will revisit work and span in more detail when we return to parallelism.

6 big-O classes

- $\mathtt{O}(1)$, or constant time
- $\mathtt{O}(log\ n)$, or logarithmic
- $\mathtt{O}(n)$, or linear
- $\mathtt{O}(n^2)$, or quadratic
- $\mathtt{O}(n^3)$, or cubic
- $\mathtt{O}(2^n), \mathtt{O}(3^n), \ldots$ exponential

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

A function is called polynomial time if it belongs to $\mathtt{O}(n^k)$ for some $k \geq 0$.

A polynomial function with highest power k is $\mathtt{O}(n^k)$. A linear function of n has the form $\alpha n + \beta$, for some constants α and β. Every linear function is $\mathtt{O}(n)$.

Every function in $\mathtt{O}(n^2)$ is also in $\mathtt{O}(n^3)$ and $\mathtt{O}(2^n)$ and $\mathtt{O}(3^n)$, but the reverse inclusions do not hold.

A function of n is called exponential time if it is $\mathtt{O}(k^n)$ for some constant k.

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

Comments

We say that "f is $\mathtt{O}(g)$". Some people use "$f = \mathtt{O}(g)$" or "$f \in \mathtt{O}(g)$".

Sometimes we'll write something like

$$f(n) = \mathtt{O}(n) + n^2$$

to mean that there is a function $g(n)$ belonging to $\mathtt{O}(n)$ such that $f(n) = g(n) + n^2$. Note that in this case it follows that f is actually $\mathtt{O}(n^2)$.

7 Common recurrences

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.

- $T(n) = T(n \texttt{ div } 2) + c$
  $T(n)$ is $\texttt{O}(log\ n)$
- $T(n) = T(n-1) + c$
  $T(n)$ is $\texttt{O}(n)$
- $T(n) = 2 * T(n \texttt{ div } 2) + c$
  $T(n)$ is $\texttt{O}(n)$
- $T(n) = T(n-1) + c * n$
  $T(n)$ is $\texttt{O}(n^2)$
- $T(n) = 2 * T(n \texttt{ div } 2) + c * n$
  $T(n)$ is $\texttt{O}(n\, log\, n)$
- $T(n) = k * T(n-1) + c$, $k > 1$
  $T(n)$ is $\texttt{O}(k^n)$

8 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.

- $T(n) = T(n \texttt{ div } 2) + c$
  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 $\texttt{O}(m)$. Extrapolating to all values of n, we expect that $T(n)$ is $\texttt{O}(log\ n)$.

- $T(n) = 2 * T(n \texttt{ div } 2) + c$
  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, $T(2^m)$ is $2^mT(0) + (2^{m-1} + \cdots + 2 + 1)c$. This suggests that $T(2^m)$ is $\texttt{O}(2^m)$. Extrapolating to all values of n, we expect that $T(n)$ is $\texttt{O}(n)$.