CSci 2041
Advanced Programming Principles
L7: Reasoning About Complexity

Eric Van Wyk

Fall 2014

Sum-to over integers

```
let rec sumTo n = match n with
  | 0 -> 0
  | n -> n + sumTo (n-1)
```

What is the complexity of this algorithm?

$O(n)$

Factorial

```
let rec fac n =
  if n = 1
  then 1
  else n * fac (n-1)
```

What is the complexity of this algorithm?

$O(n)$

List append

```
let rec append l1 l2 = match l1 with
  | [ ] -> l2
  | x::xs -> x :: (append xs l2)
```

(Called `@` in OCaml)
Here n is the length of `l1`.

What is the complexity of this algorithm?

$O(n)$

List reverse

```
let rec reverse l = match l with
  | [ ] -> [ ]
  | x::xs -> reverse xs @ [x]
```

n is the length of `l`.

What is the complexity of this algorithm?

$O(n^2)$

List reverse with an accumulator

```
let rec rev l r' = match l with
  | [] -> r'
  | (x::xs) -> rev xs (x::r')

let reverse l = rev l []
```

What is the complexity of this algorithm?

$O(n)$

```
type 'a btree = Empty
              | Tip of 'a
              | Node of ('a * 'a btree * 'a btree)

let rec insert t i = match t with
  | Empty -> Tip i
  | Tip v -> if v <= i
    then Node (i, Tip v, Tip i)
    else Node (v, Tip i, Tip v)
  | Node (v, t1, t2)
     -> if v <= i
        then Node (v, t1, insert t2 i)
 else Node (v, insert t1 i, t2)
```

```
type 'a btree = Empty
              | Tip of 'a
              | Node of ('a * 'a btree * 'a btree)

let rec find i t = match t with
  | Empty -> None
  | Tip v -> if i = v then Some i else None
  | Node(v,t1,t2) -> if v <= i
                     then find i t2
                     else find i t1
```

Complexity

Some possible questions:

- How many times is a function called?
- How many times is an operation performed?

We can state answers to these precisely as recurrence relations.
$T(n) = \ldots T(\ f(n)\ )\ldots$

We then need to convert the recurrence relation into a "closed form".
That is, one that doesn't mention T on the right hand side.

Factorial

```
let rec fac n =
  if n = 1
  then 1
  else n * fac (n-1)
```

(We assume the input n is positive, $n \geq 1$)
How many times is `fac` called?
$T(1) = 1$
$T(n) = T(n-1) + 1$
How many multiplies are there?
$T(1) = 0$
$T(n) = T(n-1) + 1$

Solving recurrence relations

We are concerned here with relatively simple recurrence relations, so we can "guess and check" to find a solution.

We can often do this by counting up and looking for a pattern.

Consider the recurrence relation for number of calls to `fac`:
$T(1) = 1$
$T(n) = T(n-1) + 1$

$T(1) = 1$
$T(2) = T(1) + 1 = 1 + 1 = 2$
$T(3) = T(2) + 1 = 2 + 1 = 3$
$T(4) = T(3) + 1 = 3 + 1 = 4$

So, perhaps $T(n) = n$.

Now check

We think $T(n) = n$, but can we check this?

Yes, with an inductive argument.
Base case: $T(1)$:
$T(1) = 1$, which matches our initial condition.

Inductive case: $T(n)$:

- $T(n)$
- $= T(n-1) + 1$, by def. of T
- $= (n-1) + 1$, by inductive hypothesis
- $= n$

Complexity

If there are n calls to `fac(n)`,

and each call takes a constant amount of time

then the complexity of `fac` is $O(n)$.

The time to execute is linear in the size of the input.

Solution to previous exercise

- The solution follows the same pattern as before.
- We are given:
  $T(1) = 0$
  $T(n) = T(n-1) + 1$
- By substitution we see a pattern:
  $T(1) = 0$
  $T(2) = T(1) + 1 = 0 + 1 = 1$
  $T(3) = T(2) + 1 = 1 + 2$
- So. guess that $T(n) = n - 1$
- Checking this by an inductive proof also follows the same pattern as before.

List append

```
let rec append l1 l2 = match l1 with
  | [ ] -> l2
  | x::xs -> x :: (append xs l2)
```

(Called `@` in OCaml)
Here n is the length of `l1`.

How many cons (`::`) applications are there?
$T(0) = 0 \qquad T(n) = T(n-1) + 1$

- Solving this is the same as our first one, with a initial condition at 0 instead 1. But the solution is the same.
- So, there are $T(n) = n$ cons operations applied for a list `l1` of length n.
- Another linear function with complexity $O(n)$.

List reverse

```
let rec reverse l = match l with
  | [ ] -> [ ]
  | x::xs -> reverse xs @ [x]
```

n is the length of `l`.
How many cons (`::`) applications are there on list of length n?
Recall, there are $T^{append}(n) = n$ cons in `@`.
$T(0) = 0$
$T(n) = T(n-1) + T^{append}(n-1)$
$T(n) = T(n-1) + n - 1$

Determine a solution

For the relation:
$T(0) = 0$
$T(n) = T(n-1) + n - 1$
We have these calculations:
$T(0) = 0$
$T(1) = T(0) + 1 - 1 = 0 + 0 = 0$
$T(2) = T(1) + 2 - 1 = 0 + 1 = 1$
$T(3) = T(2) + 3 - 1 = 1 + 2 = 3$
$T(4) = T(3) + 4 - 1 = 3 + 3 = 6$
$T(5) = T(4) + 5 - 1 = 6 + 4 = 10$
$T(6) = T(5) + 6 - 1 = 10 + 5 = 15$
$T(7) = T(6) + 7 - 1 = 15 + 6 = 21$
Hmmm.... the pattern is not so obvious...

But it isn't hard to see that on each call with list of length n we do $n - 1$ cons operations for append.

and (n-2) + (n-3) + ... + 1 cons operations for the recursive call

$$\begin{aligned}
&(n-2) + (n-3) + ... + 1\\
&= 1 + 2 + ...n - 2\\
&= (n-2)(n-1)/2\\
&= (n^2 - 2n - n + 2)/2
\end{aligned}$$

So we have:

$$\begin{aligned}
&n - 1 + (n^2 - 2n - n + 2)/2\\
&= 2(n-1)/2 + (n^2 - 2n - n + 2)/2\\
&= (2n - 2 + n^2 - 2n - n + 2)/2\\
&= (n^2 - n)/2
\end{aligned}$$

So we have a quadratic algorithm: $O(n^2)$.

List reverse with an accumulator

```
let rec rev l r' = match l with
  | [] -> r'
  | (x::xs) -> rev xs (x::r')

let reverse l = rev l []
```

How many cons operations are made?

How do solve this one?

We don't always need to solve the recurrence relation to see what the complexity of the algorithm is.

How big is the argument to the recursive call?

How many recursive calls are made on an evaluation?

How big is the value returned by the recursive call?

How is this returned value use?

Lists, ordered lists, ordered trees

Let's consider some of the functions in `ordered_list.ml` and `ordered_btree.ml` again.