Cosc 201
Algorithms and Data Structures
Lecture 6 (12/3/2025)
Analysing recursive algorithms

Brendan McCane
brendan.mccane@otago.ac.nz
Michael Albert

Induction and recursion are linked!

- When we want to analyse the time-complexity of algorithms, particularly recursive ones, the inductive approach is essential.
- It can even help with understanding nested loops or ones with complex control conditions.
- The obvious way it works in the recursive case is “The time required to do the task at size n is *some basic amount of processing, often constant* plus *the time required for one or more instances of the same task at smaller values*”
- For recurrences like the Fibonacci example, one can give upper and lower bounds on the time required and this can hopefully lead to a full analysis.

Basic Fibonacci recursion is bad

Consider the pseudocode:

Algorithm $Fib(n)$

1: **if** $n \leqslant 1$ **then**
2: **return** 1
3: **end if**
4: **return** $Fib(n-1) + Fib(n-2)$

What's the time-complexity?

Line by line analysis

Algorithm $Fib(n)$

1: **if** $n \leqslant 1$ **then**
2: **return** 1
3: **end if**
4: **return** $Fib(n-1) + Fib(n-2)$

- Line 1, always executed, some constant cost.
- Line 2, executed if $n \leqslant 1$, some constant cost.
- Line 4, executed if $n > 1$, cost equal to cost of calling $Fib(n-1)$, calling $Fib(n-2)$, and some constant cost for the addition and return.

Cost bounds

If we let $T(n)$ denote the time required for evaluating $Fib(n)$ using this algorithm, this analysis gives:

$$T(0) = T(1) = C$$
$$T(n) = D + T(n-1) + T(n-2)$$

where C and D are some positive constants.

- This certainly shows that $T(n)$ grows at least as quickly as $Fib(n)$.
- Even if we had $D = 0$ we'd get $T(n) = C \times Fib(n)$.
- In fact, the growth rates are the same – but all we really care about is that this is exponential and hence far too slow for larger values of n.

Moral: *A recursive algorithm that makes two or more recursive calls with parameter values that are close to the original will generally have exponential time complexity.*

Fixing Fibonacci

Let's adjust the pseudocode. We're thinking of *FibPair*(n) as returning the pair of numbers $(f_{n-1},\ f_n)$.

Algorithm *FibPair*(n)

1: **if** $n = 1$ **then**
2: **return** $(1,\ 1)$
3: **end if**
4: $(a,\ b) \leftarrow$ *FibPair*$(n-1)$
5: **return** $(b,\ a+b)$

Now, what's the time-complexity?

Line by line analysis

Algorithm *FibPair*(n)

1: **if** $n = 1$ **then**
2: **return** $(1, 1)$
3: **end if**
4: $(a, b) \leftarrow$ *FibPair*$(n - 1)$
5: **return** $(b, a + b)$

- Line 1, always executed, some constant cost.
- Line 2, executed if $n = 1$, some constant cost.
- Line 4, executed if $n > 1$, cost equal to cost of calling *FibPair*$(n - 1)$.
- Line 5, executed if $n > 1$, some constant cost.

Cost bounds

If we let $P(n)$ denote the time required for evaluating *FibPair*(n) using this algorithm, this analysis gives:

$$P(1) = C$$
$$P(n) = P(n-1) + D$$

where C and D are some positive constants.

Claim

$$P(n) = C + D \times (n-1)$$

Proof of claim

By induction of course!

It's true for $n = 1$ since,

$$\begin{aligned} P(1) &= C && \text{(definition of left hand side)} \\ C + D \times (1 - 1) &= C && \text{(computation of right hand side)} \end{aligned}$$

Suppose that it's true for $n - 1$. Then it's true for n as well because:

$$\begin{aligned} P(n) &= P(n-1) + D \\ &= C + D \times (n-2) + D \\ &= C + D \times (n-1) \end{aligned}$$

By induction, it's true for all $n \geqslant 1$.

Cost bounds

If we let $P(n)$ denote the time required for evaluating $FibPair(n)$ using this algorithm, this analysis gives:

$$P(1) = C$$
$$P(n) = P(n-1) + D$$

where C and D are some positive constants.

Theorem

$$P(n) = C + D \times (n-1)$$

In particular, $P(n) = \Theta(n)$.

Moral: *A recursive algorithm that makes one recursive call with a smaller parameter value and a constant amount of additional work will have at most linear time complexity.*

Correctness

How do we know that *FibPair*(n) actually produces the pair (f_{n-1}, f_n)?

By induction again of course!

It's true for $n = 1$ by design.

If it's true at $n - 1$ then the result of computing *FibPair*(n) is:

$$(f_{n-1}, f_{n-2} + f_{n-1}) = (f_{n-1}, f_n)$$

which is exactly what we want.

Correctness of recursive algorithms by induction

- How do we know that recursive algorithms work correctly?
- Because of induction!
- Find a (positive integer) *parameter* that gets smaller in all recursive calls.
- Prove inductively that “for all values of the parameter, the result computed is correct”. To do that:
  - Check correctness in all non-recursive cases.
  - Check correctness in recursive cases *assuming* correctness in the recursive calls – that’s the inductive step.
- And then we’re done.

Consider quicksort

Recall that quicksort sorts a range in an array (a group of elements between some lower index, `lo` inclusive and some upper index `hi` exclusive) as follows:

- If the length of the range (`hi - lo`) is at most 1 do nothing.
- Otherwise, choose a pivot element *p* (e.g., the element at position `lo`) and:
  - place all the items less than *p* in positions `lo` to `lo + r`
  - place all the items greater than or equal to *p* in positions `lo + r + 1` to `hi`
  - place *p* in position `lo + r`
  - call quicksort on the ranges `lo` to `lo + r` and `lo+r+1` to `hi`.

Can we be confident that works?

Quicksort works

- The parameter is just `hi - lo`, i.e., the number of elements in the range.
- This parameter gets smaller in all recursive calls because we always remove the element p so, even if it is the largest or smallest element of the range, the recursive call has a range of size at most `hi - lo - 1`.
- The non-recursive case is correct because if we have 1 or fewer elements in a range they are already sorted.
- In the recursive case, since all the elements before p are smaller than it and we assume they get sorted correctly by quicksort, and the same happens for the elements larger than p, we will get a correctly sorted array.

Divide and conquer

- Quicksort is a *divide and conquer* algorithm.
- “Large” problems are broken into smaller chunks that are handled recursively.
- Pre- and/or post-processing then converts those solutions into a solution of the large problem.
- What about our “two recursive calls is bad” moral?
- It doesn’t apply here because the sum of the sizes of the “chunks” is at most the size of the original problem.
- How this affects time-complexity is a bit subtle - we’ll address it next time as we consider *mergesort*.