CSci 2041: Advanced Programming Principles
Assessing Complexity of Programs

Gopalan Nadathur

Department of Computer Science and Engineering
University of Minnesota

Lectures in Spring 2015

What and Why?

We want to understand how to assess the costs of the functions we define

The two specific foci for the discussions

- What resources should we be counting?
- Is there a systematic way to account for them?

Some understanding of these issues is important to writing good programs

A more detailed study is something that you will carry out in an analysis of algorithms course

Asymptotic and Worst-Case Bounds

We typically want to describe how our programs do over all possible inputs

To do so, we try to quantify the relevant resource as a function of some measurement of the input

Moreover, we focus on what happens as the “size” gets large

- we want to factor out “startup noise”
- the real issues are with large inputs, for small ones we can always wait enough

A caveat: programming language characteristics can interfere with what we mean by “large inputs”

Behaviour can also be dependent on input particularities that we deal with often by looking at worst-case behaviour

Again, other possibilities exist, such as averaging over all inputs

The Big O Notation

Definition

Let f and g be two functions from $\mathcal{N}$ to the set of positive real numbers

We say that $f(n) = O(g(n))$ if there is a positive integer c and a value n_0 such that

$$\text{for every } n \geq n_0 \text{ it is the case that } f(n) \leq c * g(n)$$

Also $g(n)$ is said to be an *asymptotic upper bound* for $f(n)$

Example

Let $f(n) = 5n^3 + 2n^2 + 22n + 6 = n^3(5 + 2/n + 22/n^2 + 6/n^3)$

Then

- $f(n) = O(n^3)$ (pick $c = 6$ and $n_0 = 10$)
- Also $f(n) = O(n^4)$
- However $f(n) \neq O(n^2)$ and also $f(n) \neq O(n)$

The Ω Notation

The big O notation is useful in saying something is "no worse than" a known function

Sometimes we may also want to say that something is "no better than," for which we use the Ω notation

Definition

Let f and g be two functions from $\mathcal{N}$ to the set of positive real numbers

We say that $f(n) = \Omega(g(n))$ if there is a positive integer c and a value n_0 such that

$$\text{for every } n \geq n_0 \text{ it is the case that } c * g(n) \leq f(n)$$

Also $g(n)$ is said to be an *asymptotic lower bound* for $f(n)$

Comment: We are ignoring the "constant" but this can sometimes be the reason for preferring an implementation

Assessing a Simple OCaml Function

Consider the following function

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

What would we want to quantify with regard to this program?

- The running time as a function of n

  Depends on the cost of addition, a recursive call and a return from the recursive call

- The amount of stack space used as a function of n

  Depends on the number of recursive calls

Estimating the Running Time of Sumup

Let us quantify

- the cost of an addition as c_1
- the cost of a recursive call as c_2
- the cost of the return from a recursive call as c_3

Let $T(n)$ represent the running time of `sumup` as a function of the input n

We can express $T(n)$ in terms of $T(n-1)$ as follows

$$T(n) = \begin{cases} c_2 + c_3 & n = 0 \\ T(n-1) + c_1 + c_2 + c_3 & n > 0 \end{cases}$$

A relation of this kind which relates $T(n)$ to $T(f_i(n))$ for $0 \leq i \leq k$, is called a *recurrence relation*

By solving such recurrence relations, we can get a *closed form* expression for $T(n)$

Solving the Recurrence Relation

One possible approach is the following

- Expand the recurrence using the definition to find a pattern and hence to guess a closed form
- Check the guess using induction

For the `sumup` function

$$\begin{aligned} T(n) &= T(n-1) + c_1 + c_2 + c_3 \\ &= T(n-2) + (c_1 + c_2 + c_3) + (c_1 + c_2 + c_3) \\ &\dots \\ &= c_2 + c_3 + \underbrace{(c_1 + c_2 + c_3) + \ldots + (c_1 + c_2 + c_3)}_{n \text{ times}} \end{aligned}$$

The "guess" that we can confirm by induction:
$T(n) = c_2 + c_3 + n * (c_1 + c_2 + c_3)$

Thus, $T(n)$ is both $O(n)$ and $\Omega(n)$

Assessing the Stack Space

Let us write the stack space as a function of the input as $S(n)$

The stack space for each call of `sumup` is a fixed constant, say s

The recurrence relation for the stack space then is

$$S(n) = \begin{cases} s & n = 0 \\ S(n-1) + s & n > 0 \end{cases}$$

This has the closed form solution $S(n) = (n+1) * s$, i.e. stack space could potentially become a limitation

Some Caveats in the Assessment of Sumup

We should be careful in interpreting the solutions we have

- We are really interested in behaviour for large inputs, but the OCaml program will not work for really large n
- If we rewrite the code in a way that can deal with really large n, then the cost of addition will depend on n
- Good representations of (large) numbers are logarithmic in size, thus $O(n)$ is exponential in *input size*

In a homework problem, we will look at a sensible representation and implementation for large numbers

These kinds of issues, in general, arise less often when we deal with symbolic data

Comparison with Other Implementations

Two other implementations of `sumup`

```
let sumup' n =
  let rec sumup_helper n acc =
    match n with
    |  0 -> acc
    |  n -> sumup_helper (n - 1) (n + acc)
  in sumup_helper n 0

let sumup'' n = n * (n + 1) / 2
```

- The first function has a slightly larger stack frame but the stack does not grow
- Set up time for stack can be a little larger for first function but we save on returns
- Both functions do worse for small arguments (e.g. consider $n = 0$) but things change for larger n

Assessing the Naive Fibonnaci Function

Consider the straightforward translation of the function to OCaml code

```
let rec fib n =
  match n with
  | 1 -> 1
  | 2 -> 1
  | n -> fib (n-2) + fib (n-1)
```

A recurrence relation for the running time $T(n)$:

$$T(n) = \begin{cases} c_1 & n = 1 \\ c_1 & n = 2 \\ T(n-2) + T(n-1) + c_2 & n > 2 \end{cases}$$

Can we get a lower bound based on this recurrence relation?

A Lower Bound for the Naive Fibonnaci Function

Let us try to use the approach of expanding to guess a solution

$$\begin{aligned} T(n) &= T(n-2) + T(n-1) + c_2 \\ &= 2 * T(n-2) + T(n-3) + 2 * c_2 \\ &\geq 2 * T(n-2) + 2 * c_2 \\ &\geq 2 * T(n-4) + 2^2 * c_2 + 2 * c_2 \\ &\ldots \\ &\geq c_1 + 2^{\lceil n/2 \rceil - 1} * c_2 + \ldots + 2 * c_2 \\ &\geq 2^{\lceil n/2 \rceil} * c_2 - c_2 + c_1 \end{aligned}$$

We can now check that $T(n) \geq 2^{\lceil n/2 \rceil} * c_2 - c_2 + c_1$ by induction

From the last expression, we see that the running time is $\Omega((\sqrt{2})^n)$, an exponential lower bound

Question: How about stack space? Can we set up a recurrence relation and "solve" it?

The end result: stack space needed is O(n)

Memoization as a Programming Technique

In the naive fibonnaci function, we end up computing the same value repeatedly

An idea: Can we remember this value once and remember it for future use?

This could be a win of lookup is less expensive that re-computation

This is a technique called *memoization* and it underlies the *dynamic programming* style of algorithms

Has been used successfully in interesting problems in reducing exponential time computations to polynomial time ones

Lookup in an Association List

We can remember the number and the value of the function at that number as pairs in a list, called an *association list*

Then the following function looks up values in an association list

```
let rec lookup =
    function
      | (_,[]) -> None
      | (n,(n',v) :: memo) ->
          if (n = n') then (Some v)
          else lookup (n,memo)
```

A recurrence equation for the (worst case) time needed as a function of list size

$$T(n) = \begin{cases} c_1 & n = 0 \\ T(n) + c_2 & n > 0 \end{cases}$$

Solving this, we see that $T(n)$ is $O(n)$

Fibonnaci with Memoization

```
let fib n =
 let rec fib_memo n memo =
  match (lookup (n,memo)) with
   | None ->
      (match n with
       | 1 -> (1, (1,1) :: memo)
       | 2 -> (1, (2,1) :: memo)
       | n ->
          let (v1,memo') = (fib_memo (n-2) memo) in
          let (v2,memo'') = (fib_memo (n-1) memo') in
          let v = v1 + v2 in (v, (n,v) :: memo''))
   | (Some m) -> (m,memo)
 in let (v,_) = fib_memo n [] in v
```

A way to understand how this function works

In computing $fib(n-2) + fib(n-1)$, the first recursion builds a table that gets used in the second recursion

Assessing the Running Time of the Memoized Version

The main cost to estimate is that of *fib_memo*

We first distinguish the (worst case) running times for the two relevant cases

- Let $T(n)$ be the time needed when memo table is empty
- Let $T'(n)$ be the time needed when only the table has info

The second case involves only two lookups, i.e. $T'(n)$ is $O(n)$

Thus we have the following kind of relation for $T(n)$

$$T(n) \leq \begin{cases} c_1 & n = 1 \\ c_1 & n = 2 \\ T(n-2) + c_3 * (n-1) + c_2 & n > 2 \end{cases}$$

By expanding this recurrence, we easily get that $T(n)$ is $O(n^2)$

For fib, we can actually specialize the memo table to do better

Exponentiation by Repeated Multiplication

Given two numbers a and b, we want to calculate a^b

One way to do it is by repeated multiplication

```
let rec exp m =
   function
     | 0 -> 1
     | n ->  m * exp m (n-1)
```

The number of multiplications involved in this computation is equal to the value of the second argument

Since we need a logarithmic number of bits to represent the number, thats an *exponential* number of multiplications

Can we do better?

Exponentiation in Divide-and-Conquer Style

The key idea: Try to reduce the complexity of the problem by half at each stage

For exponentiation, we can do this at least every two multiplications to get the following code

```
let rec exp' m =
  function
  | 0 -> 1
  | 1 -> m
  | n ->
    let b = exp' m (n / 2) in
    b * b * (if n mod 2 = 0 then 1 else m)
```

Lab problem: Show that the number of multiplications for $(exp'\ m\ n)$ is less than or equal to $2 * \lceil log_2(n) \rceil$

The power of divide-and-conquer: if done right, it can cause exponential reductions in complexity

Computations on Lists

Consider the code for `append`

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

What should we be counting for this code?

- Stack space needed
- Heap space needed
- Running time for the computation

For the second question, we need to understand how data objects are stored

Once we have understood this, it is easy to see that all resource usage is $O(n)$ where n is the size of first list

Another List Example

Consider now the naive version of reverse

```
let rec reverse l =
   match l with
   | [] -> []
   | (h::t) -> append (reverse t) [h]
```

We can write the running time for *append* as $c_3 * n$ where n is the length of its first argument

Then, the running time of *reverse* as a function of the length of its argument has the following recurrence equation

$$T^{rev}(n) = \begin{cases} c_1 & n = 0 \\ T^{rev}(n-1) + c_3 * (n-1) + c_2 & n > 0 \end{cases}$$

Solving this we get $T^{rev}(n) = c_1 + c_2 * n + (c_3 * n * (n-1)/2)$

I.e. running time for *reverse* is both $O(n^2)$ and $\Omega(n^2)$

Further Questions about Reverse

- How about space usage?

  We can ignore the space used by the call to *reverse* in the body of the function

- How does it compare with the "accumulator" reverse?

```
let reverse' lst =
    let rec rev lst1 lst2 =
        match lst1 with
        | [] -> lst2
        | (h::t) -> rev t (h::lst2)
    in rev lst []
```

  The recurrence equation and solution here is similar to *sumup*

  Using accumulators can make a significant difference; here we go from a quadratic to a linear time implementation

Operations on Binary Trees

Consider searching in a binary tree

```
type 'a btree =
  | Empty
  | Node of 'a * 'a btree * 'a btree

let rec find t i =
  match t with
  | Empty -> false
  | (Node (ind,l,r)) ->
      if (i = ind) then true
      else if (i < ind) then find l i
           else find r i
```

This has the feeling of divide-and-conquer provided we can ensure that the work is divided roughly in half each time

However, is it realistic to assume this kind of "balance?"

Creating Binary Trees

The key operation in this connection is *insertion*

```
let rec insert x t =
   match t with
   | Empty -> Node (x,Empty,Empty)
   | Node (x',l,r) ->
        if (x < x') then Node (x',insert x l,r)
        else Node (x',l,insert x r)
```

The tree this function creates is dependent on the order in which we insert elements

For example, if we insert in sorted or reverse sorted order, we will actually get a “list” structure

Can we modify insertion so as to roughly preserve its dependence on tree height but still ensure the tree is balanced?

We will consider *red-black* trees towards this end

Red-Black Trees

These are binary trees where each node is coloured either red or black and that satisfy the following properties

- every node has a colour (red or black)
- leaves are (implicitly) coloured black
- if a node is coloured red, its children must be black
- every path from a given node to its leaves must have the same number of black nodes

Because of the last property, we can identify a *black height* with any given red-black tree t, written as **bh(t)**

Formally, *bh(t)* counts the number of black nodes up to, but *excluding*, the leaf

Because of the third property, there are at most $(2 * bh(t))$ nodes in any path from root to leaf in a red-black tree t

Red-Black Trees in OCaml

A type declaration for such trees

```
type color = R | B

type 'a rbtree =
    Empty
  | Node of color * 'a * 'a rbtree * 'a rbtree
```

This type declaration does not build in the invariants

However, we can define functions that check such things, e.g. consider defining the following

```
bh_RBTree : 'a rbtree -> int option

is_RBTree : 'a rbtree -> bool
is_RBTree_aux : 'a rbtree -> int * bool
```

Some Properties of Red-Black Trees

Property 1

A red-black tree with black height x has at least $2^x - 1$ data items stored in it

Proof: By induction on the tree

Base Case: Tree is *Empty*; follows from $2^0 - 1 = 0$

Inductive Step: Tree is *Node (c,i,l,r)*

The black height of l and r is at least $x - 1$, hence by the hypothesis they each have at least $2^{(x-1)} - 1$ data items

But then t itself has at least $2 * (2^{(x-1)} - 1) + 1$ data items, i.e, at least $2^x - 1$ data items

Property 2

No path from the root to leaf in a red-black tree with n items is longer than $2 * \lfloor log_2(n+1) \rfloor$

Insertion in Red-Black Trees

If we use red-black trees, we will in fact be able to exploit the divide-and-conquer paradigm

The key to realizing this advantage is defining a (reasonable) *insert* that preserves the necessary invariants

The approach to doing this:

- When we finally create a node, we colour it red; preserves all invariants except "black children for red nodes"
- We fix the problem as we go back up the tree by rotations around the parent, moving the extra red node upwards
- This could work in all cases except when we are at the root; here we can simply color the node black to finish

If this idea works, *insert* will also be $O(log_2(n))$, where n is the number of data items

Rotations to Restore Balance

Consider the following case

```
Node(B,z,Node(R,y,Node(R,x,a,b),c),d)
```

We can rotate this as follows to move the extra red to the root

```
Node(R,y,Node(B,x,a,b),Node(B,z,c,d)
```

Using the same for the other cases yields the following:

```
let balance n =
  match n with
  | ( Node(B,z,Node(R,x,Node(R,y,a,b),c),d)  |
      Node(B,z,Node(R,y,a,Node(R,x,b,c)),d)  |
      Node(B,y,a,Node(R,z,Node(R,x,b,c),d))  |
      Node(B,y,a,Node(R,x,b,Node(R,z,c,d))) ) ->
        Node(R,x,Node(B,y,a,b),Node(B,z,c,d))
  |  _ -> n
```

Completing the Definition of Insert

Using *balance*, we can define a function that does the insert correctly, except for leaving a red-red conflict at the root

```
let rec ins t d =
  match t with
  | Empty -> Node (R, d, Empty, Empty)
  | Node (c,d',l,r) ->
      if (d < d')
      then balance (Node (c,d',ins l d, r))
      else balance (Node (c,d', l,ins r d))
```

The complete *insert* function is easily defined now

```
let insert t d =
  match (ins t d) with
  | Node (_,d',l,r) -> Node (B,d',l,r)
  | _  -> (* raise an exception *)
```

Other Questions About Red-Black Trees

- What is the space cost for *insert*?
  - Stack space? $O(log_2(n)$ where n is database size
  - Heap space? also $O(log_2(n)$
- How about deletion in red-black trees?
  - The idea: Proceed as before, this time using rotations to equalize black heights if a black node was deleted
  - Also, in the functional implementation, deletion is often obviated
- The code for red-black trees will be made available in the code directory

Summarizing the Discussions about Complexity

- Good programming involves understanding costs of programs; some solutions can be really bad
- The kinds of costs we want to consider
  - The running time of the function
  - The stack space required
  - The heap space—we have also considered how to assess these costs

  An important related issue: What should we be using as a parameter in assessing these costs?
- Some simple programming techniques that we might consider in getting better behaviour

  using accumulators, memoization, divide-and-conquer