Mat 2345 — Discrete Math

Week 12

Fall 2013

Student Responsibilities — Week 12

- **Reading**: Textbook, Section 4.4 & 4.5
- **Assignments**:

  Sec 4.4 8, 10, 24, 28
  Sec 4.5 2, 4, 7, 12
- **Attendance**: Frostily Encouraged

Week 12 Overview

- Sec 4.4. Recursive Algorithms
- Sec 4.5. Program Correctness

Section 4.4 Recursive Algorithms

A recursive procedure to find the max in a non–empty list.

We will assume we have built–in functions:

- `Length()` — which returns the number of elements in the list
- `Max()` — which returns the larger of two values
- `Listhead()` — which returns the first element in a list

Note: `Max()` requires one comparison

```
procedure Maxlist(...list...){
// PRE:  list is not empty
// POST: returns the largest element in list
// strip off list head and pass on the remainder

if Length(list) is 1 then
    return Listhead(list)
else
    return Max(Listhead(list),
               Maxlist(remainder_of_list))
}
```

What happens with the list $\{29\}$?

With the list $\{3, 8, 5\}$?

How Many Comparisons?

The recurrence equation for the number of comparisons required for a list of length n, $C(n)$ is:

$$\begin{array}{rcll} C(1) & = & 0 & \text{the initial condition} \\ C(n) & = & 1 + C(n-1) & \text{the recurrence equation} \\ \text{So, } C(n) & \in & O(n) & \text{as we would expect} \end{array}$$

A Variant of `Maxlist()`

Assuming the list length is a power of 2, here is a variant of `Maxlist()` using a Divide–and–Conquer approach.

- Divide the list in half, and find the maximum of each half
- Find the `Max()` of the maximum of the two halves
- Apply these steps to each list half recursively.
- What could the base case(s) be?

Maxlist2() Algorithm

```
procedure Maxlist2(...list...){
// PRE:  list is not empty
// POST: returns the largest element in list
// Divide list into two lists, take the max of
//   the two halves (recursively)

if Length(list) is 1 then
     return Listhead(list)
else
     a = Maxlist2(first half of list)
     b = Maxlist2(second half of list)
     return Max(a, b)
}
```

What happens with the list $\{29, 7\}$?

With the list $\{3, 8, 5, 7\}$?

How Many Comparisons in Maxlist2()?

- There are two calls to Maxlist2(), each of which requires $C(\frac{n}{2})$ operations to find maximum.
- One comparison is required by the Max() function

The **recurrence equation** for the number of comparisons required for a list of length n, $C(n)$, is:

$$\begin{array}{lcll} C(1) & = & 0 & \text{the initial condition} \\ c(n) & = & 2C(\frac{n}{2}) + 1 & \text{the recurrence equation} \end{array}$$

Consider A Sampling

n	$C(n) = 2C(\frac{n}{2}) + 1$
$2^0 = 1$	$1 = 2^1 - 1$
$2^1 = 2$	$3 = 2^2 - 1$
$2^2 = 4$	$7 = 2^3 - 1$
$2^3 = 8$	$15 = 2^4 - 1$
$2^4 = 16$	$31 = 2^5 - 1$
$\vdots$	$\vdots$
$2^{\log n} = n$	$2^{\log(n)+1} - 1 = 2n - 1 \in O(n)$

Thus, $C(n) = 2^{\log(n)+1} - 1 \in O(n)$

Practice I: **Prove** $3n^2 + 5n + 4 \in O(n^2)$

Definition of Big–Oh: $f(n) \in O(g(n))$ if there exists positive constants c and N_0 such that $\forall\ n \geq N_0$ we have $f(n) \leq cg(n)$

We need to find $c > 0$ and $N_0 > 0$ such that:

$$3n^2 + 5n + 4 \leq cn^2 \quad \forall n \geq N_0$$

We note that

$$\begin{array}{ll} 3n^2 + 5n + 4 & \leq 3n^2 + 5n^2 + 4n^2, \text{ when } n > 0 \\ & \leq 12n^2 \end{array}$$

and we can choose $c = 12$

Practice I, Cont.

To find N_0:
$$\begin{array}{rcl} 3n^2 + 5n + 4 & = & 12n^2 \\ 0 & = & 9n^2 - 5n - 4 \end{array}$$

when $n = 1$, $9(1)^2 - 5(1) - 4 = 9 - 5 - 4 = 0$

Thus, $3n^2 + 5n + 4 \leq 12n^2 \ \ \forall\ n \geq 1$, and therefore, $3n^2 + 5n + 4 \in O(n^2)$

Practice II.
Given $T(n) = 2n - 1$, prove that $T(n) \in O(n)$

Practice III.
Prove that $T(n) = 3n + 2$ if

$$T(n) = \begin{cases} 2 & n = 0 \\ 3 + T(n-1) & n > 0 \end{cases}$$

MergeSort Algorithm

```
list MergeSort(list[1..n]){
// PRE: none
// POST: returns list[1..n] in sorted order
// Functional dependency:  Merge()

  if n is 0
     return an empty list
  else if n is 1
     return list[1]
  else {
    list A = MergeSort(list[1..n/2])
    list B = MergeSort(list[n/2 + 1..n])
    list C = Merge(A, B)
    return C
  }
}
```

Time Complexity of `MergeSort()`

Prove by induction that the time complexity of `MergeSort()`, $T(n) \in O(n \log n)$

What we need to do:

- Establish a Base Case for some small n
- Prove $T(k) \leq cf(k) \rightarrow T(2k) \leq cf(2k)$

In particular, we need to prove $\forall k \geq N_0$ that:

$$\begin{aligned} T(k) \leq ck\log k \quad \rightarrow \quad T(2k) &\leq c2k\log(2k) \\ &= c2k(\log 2 + \log k) \\ &= c2k\log k + c2k \end{aligned}$$

where $k = 2^m$ for some $m \geq 0$, *wlog**

* *without loss of generality*

Base Case

Let $n = 1$.
$n\log n = (1)\log(1) = 1(0) = 0$
But, $T(n)$ is always positive, so this is not a good base case.
Try a larger number.

Let $n = 2$.
$T(2) =$ Time to divide
+ time to MergeSort halves
+ time to Merge
$= 1 + 1 + 1 + 2 = 5$

while $n \log n = 2\log 2 = 2(1) = 2$

Can we find a constant $c > 0$ such that $5 \leq 2c$?

$\frac{5}{2} \leq c$, so $\frac{5}{2}$ is a lower bound on c

Inductive Hypothesis

Assume for some arbitrary $k \geq 2$ that $T(k) \leq ck\log k$

Inductive Step — **Show** $T(2k) \leq 2ck\log k + 2ck$

$$\begin{aligned} T(2k) &\leq 1 + T(\lceil \tfrac{2k}{2} \rceil) + T(\lfloor \tfrac{2k}{2} \rfloor) + 2k \\ &\leq T(k) + T(k) + 2k + 1 \\ &\leq 2T(k) + 2k + 1 \\ &\leq 2(ck\log k) + 2k + 1 \\ &\leq 2ck\log k + 2k + 1 \end{aligned}$$

Inductive Step, Cont.

Now, can we find a c such that

$$\begin{aligned} 2ck\log k + 2k + 1 &\leq 2ck\log k + 2ck \\ 2k + 1 &\leq 2ck \\ 1 &\leq 2ck - 2k \\ 1 &\leq 2k(c-1) \end{aligned}$$

Since $k \geq 2$ from base case, $(c-1) \geq \frac{1}{4}$ or $c \geq \frac{5}{4}$
We had a lower bound of $\frac{5}{2}$, so we can choose $c = 3$.

Thus, $T(n) \in O(n\log n) \quad \forall\, n \geq 2$.

More on Complexity

- If an algorithm is composed of several parts, then its time complexity is the **sum** of the complexities of its parts.
- We must be able to **evaluate** these **summations**.
- Things become even more complicated when the algorithm contains loops, each iteration of which is a different complexity.

An Example: Suppose $S_n = \sum_{i=1}^{n} i^2$

- We saw $\sum_{i=1}^{n} i = \frac{n(n+1)}{2} = \frac{(n^2+n)}{2} \leq n^2$, and is, in fact, $\Theta(n^2)$
- So, we guess $\sum_{i=1}^{n} i^2 \leq \sum_{i=1}^{n} n^2 = n^3$. Maybe $S_n \in \Theta(n^3)$.
- We can prove our guess correct, and find the minimum constant of difference between S_n and n^3 by induction:
- Guess: $\sum_{i=1}^{n} i^2 = an^3 + bn^2 + cn + d = P(n)$
- Notice that $\sum_{i=1}^{n+1} i^2 - \sum_{i=1}^{n} i^2 = (n+1)^2$

So, $P(n+1) = P(n) + (n+1)^2$

Thus,

$$a(n+1)^3 + b(n+1)^2 + c(n+1) + d = an^3 + bn^2 + cn + d + (n+1)^2$$

$$a(n^3 + 3n^2 + 3n + 1) + b(n^2 + 2n + 1) + cn + c + d = an^3 + bn^2 + cn + d + n^2 + 2n + 1$$

$$an^3 + 3an^2 + 3an + a + bn^2 + 2bn + b + cn + c + d = an^3 + bn^2 + cn + d + n^2 + 2n + 1$$

Hence:

$$\begin{aligned} 3an^2 + 3an + a + 2bn + b + c &= n^2 + 2n + 1, \quad \text{or} \\ 3an^2 + (3a + 2b)n + (a + b + c) &= n^2 + 2n + 1 \end{aligned}$$

Since coefficients of the same power of n must be equal:

$$\begin{aligned} 3a &= 1 \\ a &= \tfrac{1}{3} \end{aligned} \qquad \begin{aligned} (3a+2b) &= 2 \\ 3(\tfrac{1}{3}) + 2b &= 2 \\ 2b &= 1 \\ b &= \tfrac{1}{2} \end{aligned} \qquad \begin{aligned} a + b + c &= 1 \\ \tfrac{1}{3} + \tfrac{1}{2} + c &= 1 \\ c &= 1 - \tfrac{1}{3} - \tfrac{1}{2} \\ c &= \tfrac{1}{6} \end{aligned}$$

And we can choose $d = 0$

Hence,

$$\begin{aligned} P(n) &= \tfrac{1}{3}(n^3) + \tfrac{1}{2}(n^2) + \tfrac{1}{6}(n) \\ &= \tfrac{2}{6}(n^3) + \tfrac{3}{6}(n^2) + \tfrac{1}{6}(n) \\ &= \frac{(2n^3+3n^2+n)}{6} \\ &= \frac{n(2n^2+3n+1)}{6} \\ &= \frac{n(n+1)(2n+1)}{6} \end{aligned}$$

Now, we wish to prove $S_n \in \Theta(n^3)$, or, that S_n is a third degree polynomial, by induction.

Base Case

Let n = 1

lhs: $\sum_{i=1}^{1} i^2 = 1^2 = 1$

rhs: $P(1) = \frac{1(1+1)(2(1)+1)}{6} = \frac{1(2)(3)}{6} = \frac{6}{6} = 1 \quad \checkmark$

Inductive Hypothesis

Assume for some arbitrary $k \geq 1$, that $S_k = P(k)$

That is, $\sum_{i=1}^{k} i^2 = \frac{k(k+1)(2k+1)}{6}$

Inductive Step

Show $S_{k+1} = P(k+1) = \frac{(k+1)(k+2)(2k+3)}{6}$

$$
\begin{array}{lllll}
\text{lhs} = S_{k+1} & = & S_k + (k+1)^2 & & \text{defn of } \sum \\
 & = & \frac{k(k+1)(2k+1)}{6} + (k+1)^2 & & \text{IH \& subst.} \\
 & = & \frac{k(k+1)(2k+1) + 6(k+1)^2}{6} & & \text{Alg. Man.} \\
 & = & \frac{(k+1)[k(2k+1) + 6(k+1)]}{6} & & \text{Alg. Man.} \\
 & = & \frac{(k+1)(2k^2+k+6k+6)}{6} & & \text{Alg. Man.} \\
 & = & \frac{(k+1)(2k^2+7k+6)}{6} & & \text{Alg. Man.} \\
 & = & \frac{(k+1)(k+2)(2k+3)}{6} = rhs & \surd & \text{Alg. Man.}
\end{array}
$$

Thus, $S_n = P(n) \ \forall\, n \geq 1$

Hence, $S_n \in \Theta(n^3)$

Section 4.5 — Program Correctness

- A brief introduction to the area of program verification, tying together the **rules of logic**, **proof techniques**, and the concept of an **algorithm**.
- **Program verification** means to prove the correctness of the program.
- Why is this important? Why can't we merely run testcases?
- A program is said to be **correct** if it produces the correct output for every possible input.

Correctness Proof

A correctness proof for a program consists of **two parts**:

1. Establish the **partial correctness** of the program.
   If the program terminates, then it halts with the correct answer.
2. Show that the program **always terminates**.

Proving Output Correct

We need two propositions to determine what is meant by **produce the correct output**.

1. **Initial Assertion**: the properties the input values must have. (p)
2. **Final Assertion**: the properties the output of the program should have if the program did what was intended. (q)

A program segment S is said to be
partially correct with respect to p **and** q, $[p\ \{S\}\ q]$,
if — whenever p is TRUE for the input values of S
and S terminates, —
then q is TRUE for the output values of S.

Example

```
p :  x = 1        // initial assertion

     y = 2        // segment
     z = x + y    //    S

q :  z = 3        // final assertion
```

Is $[p\ \{S\}\ q]$ TRUE**?**

Composition Rule: $[p\ \{S_1\}\ q]$ and $[q\ \{S_2\}\ r] \rightarrow [p\ \{S_1; S_2\}\ r]$

Rules of Inference: Conditional Statements

IF condition THEN block

BLOCK is executed when **condition** is TRUE, and it is not executed when **condition** is FALSE.

To verify correctness with respect to p and q, we must show:

1. When p is TRUE and **condition** is also TRUE, then q is TRUE after BLOCK terminates.
2. When p is TRUE and **condition** is FALSE, q is TRUE (since BLOCK does not execute.

This leads to the following rule of inference:

$$[(p \wedge condition)\{block\}q \quad \text{and} \quad (p \wedge \neg condition) \rightarrow q] \rightarrow p\{if\ \ condition\ then\ block\}q$$

Example

```
p : none

    if x > y then y = x    // Segment S

q : y >= x
```

Is $[p\ \{S\}\ q]$ **TRUE?**

IF...THEN...ELSE Statements

IF condition THEN block1 ELSE block2

If **condition** is TRUE, then `block1` executes;
if **condition** is FALSE, then `block2` executes.

To verify correctness with respect to p and q, we must show:

1. When p is TRUE and **condition** is also TRUE, then q is TRUE after BLOCK1 terminates.
2. When p is TRUE and **condition** is FALSE, q is TRUE after BLOCK2 terminates.

This leads to the following rule of inference:

$$[(p \wedge condition)\{block1\}q \quad \text{and} \quad (p \wedge \neg condition)\{block2\}q] \rightarrow p\{if\ \ condition\ then\ block1\ else\ block2\}q$$

Example

```
p : none

    if  x < 0  then  abs = -x    // Segment S
               else  abs =  x

q : abs = |x|
```

Is $[p\ \{S\}\ q]$ **TRUE?**
I.e., is the segment correct?

Loop Invariants — While Loops

WHILE condition block

Where BLOCK is repeatedly executed until `condition` becomes FALSE.

Loop Invariant: an assertion that remains TRUE **each** time BLOCK is executed.

I.e., p is a loop invariant if $(p \wedge condition)\{block\}p$ is TRUE

Let p be a loop invariant.

If p is TRUE before Segment `S` is executed, then p and ¬`condition` are TRUE after the loop terminates (if it does).

Hence: $(p \wedge condition)\{S\}p$

$$\therefore \quad p\{while\ condition\ S\}(\neg condition \wedge p)$$

Example

We wish to verify the following code segment terminates with `factorial = n!` when `n` is a positive integer.

Our loop invariant p is: **factorial = i! and i** $\leq$ **n**

```
i = 1
factorial = 1
while i < n  {
   i = i + 1
   factorial = factorial * i
}
```

[Base Case] p is TRUE before we enter the loop since `factorial = 1 = 1!`, and $1 \leq n$.

[Inductive Hypothesis] Assume for some arbitrary $k \geq 1$ that p is TRUE. Thus $i < k$ (so we enter the loop again), and `factorial= (i-1)!`.

[Inductive Step] Show p is still TRUE after execution of the loop. Thus $i \leq k$ and `factorial = i!`.

First, i is incremented by 1

Thus $i \leq k$ since we assumed $i < k$, and i and $k \geq 1$.

Also, `factorial`, which was $(i-1)!$ by IH, is set to $(i-1)! * i = i!$

Hence, p remains true.

Since p remains TRUE, p is a loop invariant and thus the assertion:

$$[p \wedge (i < n)]\{S\}p \quad \text{is TRUE}$$

It follows that the assertion:

$$p\{while\ i < n\ \ S\}[(i \geq n) \wedge p] \text{ is also true.}$$

Furthermore, the loop terminates after $n-1$ iterations with $i = n$, since:

1. i is assigned the value 1 at the beginning of the program,
2. 1 is added to i during each iteration of the loop, and
3. the loop terminates when $i \geq n$

Thus, at termination, `factorial = n!`.

We split larger segments of code into component parts, and use the rule of composition to build the correctness proof.

$$(p = p_1)\{S_1\}q_1, \quad q_1\{S_2\}q_2, \quad \ldots, \quad q_{n-1}\{S_n\}(q_n = q) \ \rightarrow \ p\{S_1; S_2; \ldots; S_n\}q$$