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:

$$
C(1) = 0 \quad \text{the initial condition}
$$

$$
C(n) = 1 + C(n - 1) \quad \text{the recurrence equation}
$$

So, $C(n) \in O(n)$ as we would expect
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 $C\left(\frac{n}{2}\right)$ 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:

\[
C(1) = 0 \quad \text{the initial condition}
\]

\[
c(n) = 2C\left(\frac{n}{2}\right) + 1 \quad \text{the recurrence equation}
\]
Consider A Sampling

\[
C(n) = 2C\left(\frac{n}{2}\right) + 1
\]

<table>
<thead>
<tr>
<th>(n)</th>
<th>(C(n) = 2^\log(n) + 1 - 1)</th>
</tr>
</thead>
<tbody>
<tr>
<td>(2^0) = 1</td>
<td>1 = (2^1 - 1)</td>
</tr>
<tr>
<td>(2^1) = 2</td>
<td>3 = (2^2 - 1)</td>
</tr>
<tr>
<td>(2^2) = 4</td>
<td>7 = (2^3 - 1)</td>
</tr>
<tr>
<td>(2^3) = 8</td>
<td>15 = (2^4 - 1)</td>
</tr>
<tr>
<td>(2^4) = 16</td>
<td>31 = (2^5 - 1)</td>
</tr>
<tr>
<td>(\vdots)</td>
<td>(\vdots)</td>
</tr>
<tr>
<td>(2^\log n) = (n)</td>
<td>(2^\log(n) + 1 - 1 = 2n - 1 \in O(n))</td>
</tr>
</tbody>
</table>

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:

\[
3n^2 + 5n + 4 \leq cn^2 \ \forall n \geq N_0
\]

We note that

\[
3n^2 + 5n + 4 \leq 3n^2 + 5n^2 + 4n^2, \text{ when } n > 0
\]

\[
\leq 12n^2
\]

and we can choose \(c = 12\)
To find $N_0$: \[ 3n^2 + 5n + 4 = 12n^2 \]
\[ 0 = 9n^2 - 5n - 4 \]

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

```python
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:

$$T(k) \leq ck \log k \rightarrow T(2k) \leq c2k \log(2k)$$

$$= c2k(\log 2 + \log k)$$

$$= c2k \log k + c2k$$

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

*without loss of generality
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 $n = 2$.

$T(2) = \text{Time to divide} + \text{time to MergeSort halves} + \text{time to Merge}$

$= 1 + 1 + 1 + 2 = 5$

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

\[ T(2k) \leq 2ck \log k + 2ck \]

\[
T(2k) \leq 1 + T(\lceil \frac{2k}{2} \rceil) + T(\lfloor \frac{2k}{2} \rfloor) + 2k \\
\leq T(k) + T(k) + 2k + 1 \\
\leq 2T(k) + 2k + 1 \\
\leq 2(ck \log k) + 2k + 1 \\
\leq 2ck \log k + 2k + 1
\]
Inductive Step, Cont.

Now, can we find a $c$ such that

$$2ck \log k + 2k + 1 \leq 2ck \log k + 2ck$$

$$2k + 1 \leq 2ck$$

$$1 \leq 2ck - 2k$$

$$1 \leq 2k(c - 1)$$

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:

\[ 3an^2 + 3an + a + 2bn + b + c \quad = \quad n^2 + 2n + 1, \quad \text{or} \]
\[ 3an^2 + (3a + 2b)n + (a + b + c) \quad = \quad n^2 + 2n + 1 \]

Since coefficients of the same power of \( n \) must be equal:

\[
\begin{align*}
3a &= 1 & (3a+2b) &= 2 & a + b + c &= 1 \\
a &= \frac{1}{3} & 3\left(\frac{1}{3}\right) + 2b &= 2 & \frac{1}{3} + \frac{1}{2} + c &= 1 \\
2b &= 1 & c &= 1 - \frac{1}{3} - \frac{1}{2} \\
b &= \frac{1}{2} & c &= \frac{1}{6}
\end{align*}
\]

And we can choose \( d = 0 \)
Hence,

\[ P(n) = \frac{1}{3} (n^3) + \frac{1}{2} (n^2) + \frac{1}{6} (n) \]

\[ = \frac{2}{6} (n^3) + \frac{3}{6} (n^2) + \frac{1}{6} (n) \]

\[ = \frac{(2n^3+3n^2+n)}{6} \]

\[ = \frac{n(2n^2+3n+1)}{6} \]

\[ = \frac{n(n+1)(2n+1)}{6} \]

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

lhs = $S_{k+1} = S_k + (k + 1)^2$

= $\frac{k(k+1)(2k+1)}{6} + (k + 1)^2$  

IH & subst.

= $\frac{k(k+1)(2k+1) + 6(k+1)^2}{6}$  

Alg. Man.

= $\frac{(k+1)[k(2k+1) + 6(k+1)]}{6}$  

Alg. Man.

= $\frac{(k+1)(2k^2+k+6k+6)}{6}$  

Alg. Man.

= $\frac{(k+1)(2k^2+7k+6)}{6}$  

Alg. Man.

= $\frac{(k+1)(k+2)(2k+3)}{6} = rhs \sqrt{\text{Alg. Man.}}$

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.
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

\[ p : \ x = 1 \quad /\ / \text{ initial assertion} \]

\[ y = 2 \quad /\ / \text{ segment} \]

\[ z = x + y \quad /\ S \]

\[ q : \ z = 3 \quad /\ / \text{ final assertion} \]

Is \( [p \{ S \} q] \) TRUE?

**Composition Rule:** \( [p \{ S_1 \} q] \text{ and } [q \{ S_2 \} r] \rightarrow [p \{ S_1; S_2 \} r] \)
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:

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

\[ p : \text{none} \]

if \( x > y \) then \( y = x \)  // Segment \( S \)

\[ q : y \geq x \]

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:

\[
\left[(p \land condition)\{\text{block1}\}q \quad \text{and} \quad (p \land \neg condition)\{\text{block2}\}q \right] 
\rightarrow p\{\text{if condition then block1 else block2}\}q
\]
Example

\[ p : \text{none} \]

\[
\begin{align*}
\text{if } x < 0 \text{ then } \text{abs} &= -x & \text{// Segment } S \\
\text{else } \text{abs} &= x
\end{align*}
\]

\[ q : \text{abs} = |x| \]

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 $(p \land \text{condition})\{\text{block}\}p$ is TRUE.

Let $p$ be a loop invariant.

If $p$ is TRUE before Segment $S$ is executed, then $p$ and $\neg\text{condition}$ are TRUE after the loop terminates (if it does).

Hence: $(p \land \text{condition})\{S\}p$

$\therefore p\{\text{while condition } S\}(\neg\text{condition} \land p)$
Example

We wish to verify the following code segment terminates with \( \text{factorial} = n! \) when \( n \) is a positive integer.

Our loop invariant \( p \) is: \( \text{factorial} = i! \) and \( i \leq n \)

\[
\begin{align*}
&\text{i = 1} \\
&\text{factorial = 1} \\
&\text{while i < n \{ } \\
&\quad \text{i = i + 1} \\
&\quad \text{factorial = factorial * i} \\
&\text{\}}
\end{align*}
\]
**Base Case**  $p$ is $\text{TRUE}$ before we enter the loop since $\text{factorial} = 1 = 1!$, and $1 \leq n$.

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

**Inductive Step** Show $p$ is still $\text{TRUE}$ after execution of the loop. Thus $i \leq k$ and $\text{factorial} = i!$. 
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 $(i - 1)! \ast i = i!$

Hence, $p$ remains true.

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

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

It follows that the assertion:

$$p\{\text{while } i < n \ S\}[(i \geq n) \land p]$$

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, \( \text{factorial} = n! \).

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

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