CSC236 tutorial exercises, Week #7
sample solutions

1. For a given string s, we'll say that s' is a **top-level parenthesized substring** ('tlps') of s iff the following conditions are met:

   i. s' is a substring of s

   ii. s' is of the form (q), where q is a (possibly empty) string with balanced parentheses

   iii. there does not exist any longer string, s'', such that s' is contained within s'' and s'' satisfies the above 2 conditions

   For example, the string $s =$ '((hello))hi((a)(b)(c))' has two tlps's: '((hello))' and '((a)(b)(c))'. On the other hand

   - '(a)' is not a tlps, because it is contained within the tlps '((a)(b)(c))'
   - '((hello))hi' is not a tlps, because it is not enclosed in parentheses
   - s itself is not a tlps because the parentheses in '(hello))hi((a)(b)(c)' are not balanced

```
def extract_tlps(s):
    """Pre: s is a string, and its parentheses are balanced.
    Post: Return a list of all the tlps's in s

    >>> extract_tlps('sorry (not sorry)')
    ['(not sorry)']
    >>> extract_tlps('((hello))hi((a)(b)(c))')
    ['((hello))', '((a)(b)(c))']
    """
    R = []
    l = 0
    par = ''
    i = 0
    while i < len(s):
        c = s[i]
        if c == '(':
            l += 1
        if l > 0:
            par += c
        if c == ')':
            l -= 1
            if l == 0:
                R.append(par)
                par = ''
        i += 1
    return R
```

(a) Devise a loop invariant for this function which is sufficient to prove partial correctness. Do not attempt to prove the invariant.[1]

Solution:
At the end of each iteration j, R_j contains every tlps of s which is contained within $s[:i_j]$.

Remark: If we were being very careful, we might also add the invariant "i_j is an integer", since the slice expression $s[:i_j]$ is not well-defined otherwise.

(b) Assume the invariant you wrote in part (a) is true, and use it to prove the partial correctness of `extract_tlps` (i.e. if `extract_tlps(s)` terminates, then the postcondition is satisfied).

Solution:
Let s be an arbitrary string satisfying the precondition, and assume that the while loop exits after some iteration, call it j.

By our invariant, R_j contains every tlps in $s[:i_j]$. By the loop condition, we know that $i_j \geq \text{len}(s)$, which means that $s[:i_j] = s$. So our return value R_j contains every tlps in s, as required by the postcondition. ■

(c) In the above parts, you may have found that a simple invariant was sufficient to prove partial correctness. However, proving that invariant directly by induction may prove very difficult. In this case, we need to strengthen the induction, by adding additional invariants which help us prove the main one. Here are two useful examples:

i. l_j is the 'left surplus' of par_j, i.e. the count of left parentheses in par_j minus the count of right parentheses in par_j

ii. l_j is the left surplus of $s[:i_j]$.

Use induction to prove these invariants.

Hint: You may use assume that any prefix of a string with balanced parentheses has at least as many left parentheses as right parentheses. (We proved essentially this fact in lecture when talking about structural induction.)

Solution:
$l_0 = 0$, and par_0 and $s[:i_0]$ are both empty strings, so our invariant trivially holds before the first iteration.

Assume the invariant holds at the end of some iteration j.

First, observe that $l_j \geq 0$, by 1(c)ii, since the precondition says that s has balanced parentheses, and any prefix of a string with balanced parentheses must have a non-negative left surplus.

I will prove the invariant holds at the end of the $j+1$th iteration (assuming it occurs), by cases according to the value of $c_{j+1} = s[i_j]$, i.e. the character inspected in this iteration.

Case 1: $c_{j+1} = ($. By line 17, $l_{j+1} = l_j + 1$. This is matched by a corresponding increase in the left parenthesis count of the two strings we're interested in:

- $i_{j+1} = i_j + 1$, so the prefix of s referred to in 1(c)ii is grown by one character, which is c_{j+1}, a left paren.
- $par_{j+1} = par_j + ($, by line 19. The if is satisfied, since $l_j \geq 0$, meaning that $l_{j+1} = l_j + 1 > 0$.

[1] Even though you don't need to prove it, your invariant should be *true*. For example, "R_j contains all tlps's in s" or "$i_j = i_j + 1$" are not appropriate invariants, even though either would technically be sufficient for proving partial correctness.

Case 2: $c_{j+1} =)$. This implies $l_j > 0$. If l_j were allowed to be zero, it would mean that $s[:i_{j+1}]$ has more right parens than left, contradicting the precondition. Since $l_j > 0$, we add the right paren to par by line 19. Then we enter the 'if' body after line 20. And set $l_{j+1} = l_j - 1$, satisfying 1(c)ii. From here we must consider two subcases:

Subcase 2a: $l_j = 1$. Then $l_{j+1} = 0$ and par_{j+1} is the empty string, satisfying 1(c)i.

Subcase 2b: $l_j > 1$. Then par_{j+1} is $par_j+)$ (by line 19), which agrees with the decrease in l, satisfying 1(c)i.

Case 3: c_{j+1} is a non-parenthesis character. Then $l_{j+1} = l_j$, and the left-surplus of both strings under consideration (par_{j+1} and $s[:i_{j+1}]$ is unchanged from the jth iteration, satisfying the invariants.

(d) Is the invariant from part (c) enough to prove your original invariant from part (a)? Brainstorm additional invariants which could be used to complete the proof. You do not need to prove your final set of invariants (the full proof would be fairly long), but you should have some idea of what the overall structure of the proof would look like.

Solution:

In addition to our invariant from part (a) and the one we proved in part (c), we add the following invariant:

- if there exists a tlps $s' = s[a:b]$ such that $a < i_j$ and R_j does not contain s', then $par_j = s'[a:i]$. Otherwise, par_j is empty.
  - Less formally, this says that if we're currently somewhere in the middle of a tlps, then par_j contains the prefix of that tlps that we've seen so far.

Note: The explanation that follows is not a required part of the solution for part (d), but it's presented for the interest of the reader.

This should be enough to prove our main invariant from (a). To do so, we basically need to show that we append to R iff we're at the end of a tlps. Here's a brief proof sketch:

By the code, we can show that the append is reached in the $j+1$th iteration iff $l_j = 1$, and $s[i_j] =$ ')'. By 1(c)i, this means par_j is non-empty and has 1 more left paren than right. By our new invariant, par_j is a prefix of a tlps. Thus, by finding a right paren in the $j+1$th iteration, we complete the tlps, and append it to R_j.

What about proving the new invariant? There are a few cases to consider. Here's one natural breakdown:

- If $s[i_j]$ is the start of a tlps, then by definition it must be a left paren. So by the code, we append it to par_j (which must be empty by the IH, since by definition, tlps's are non-overlapping), as required.
- If $s[i_j]$ is the last character of a tlps, then we can argue that line 24 is reached and par_{j+1} is the empty string (which is correct, since there is now no tlps with a starting point $< i_{j+1}$ which is not in R_{j+1}).
- If $s[i_j]$ is somewhere in the middle of a tlps, then we append it to par_j, since $l_j > 0$, by part ii. of the definition of tlps.
- If $s[i_j]$ is not part of a tlps, we need to show that $par_{j+1} = par_j =$ the empty string. This follows from the 'main' (part (a)) invariant in our inductive hypothesis, plus the fact that all tlps's have length at least 2 (by part ii. of their definition).

Note: Your invariant should bear some resemblance to the one given above, though it's possible to present the same idea in different terms. For example, here's a more whimsical formulation: par_j is the longest string which is

- a suffix of $s[:i_j]$
- a prefix of a tlps

2. Consider the function `bitcount` defined below:

```
def bitcount(n):
    """Pre: n is a positive int.
    Post: return the number of digits in the binary representation of n
    """
    i = 1
    while n > 1:
        n = n//2
        i += 1
    return i
```

Prove that `bitcount` is partially correct with respect to the specification in the docstring. You will need to prove an appropriate loop invariant, then show that it implies the postcondition given that the loop exits.

You may assume that for $n \in \mathbb{N}^+$, n has k bits iff 2^{k-1} is the largest power of 2 which is less than or equal to n. Or, equivalently, $k = \lfloor \log n \rfloor + 1$.

Solution:
Before proving this function correct, we should try to convince ourselves that it's correct, and develop some intuition for how it works. The gist of it seems to be that we repeatedly divide n by 2 (rounding down), until we get to 1. Then we return the number of divisions we did plus one. This sounds closely related to the base-2 logarithm of n, which makes sense.

Looking at the binary representation of n as the program executes, I see a pattern. For example, let $n = 13 = 1101_2$. The sequence of values taken on by n, starting from n_0 is: $1101_2, 110_2, 11_2, 1_2$. Each iteration of the loop truncates the rightmost digit in n's binary representation. (If you're familiar with a C-like programming language, you may recognize this as a 'right shift'.) I will prove a somewhat weaker version of this in my invariant.

For convenience, let $\mathrm{b}(n)$ denote the number of bits in the binary representation of n.

Lemma 0.1. *For all $n > 1$, $\mathrm{b}(n \,//\, 2) = \mathrm{b}(n) - 1$*

Proof. Let $n \in \mathbb{N}$ be greater than 1 with some number of bits k. By the fact given in the question, 2^{k-1} is the largest power of $2 \leq n$. Thus we can write n as $2^{k-1} + r$, where $r < 2^{k-1}$.

Thus

$$\begin{aligned} n \,//\, 2 &= (2^{k-1} + r) \,//\, 2 \\ &= 2^{k-2} + r \,//\, 2 \qquad \# \; k - 1 > 0, \text{ since } n > 1 \end{aligned}$$

Note that $r \,//\, 2 \leq r/2 < 2^{k-2}$. Thus, 2^{k-2} is the largest power of 2 less than or equal to $n \,//\, 2$, meaning $n \,//\, 2$ has $k - 1$ bits. □

We are now ready to prove the following loop invariant.

Lemma 0.2 (Loop invariant). *At the end of each iteration j,*

(a) $\mathrm{b}(n_j) = \mathrm{b}(n_0) - i_j + 1$

(b) $n_j \in \mathbb{N}^+$

Proof. $i_0 = 1$, so $\mathrm{b}(n_0) = \mathrm{b}(n_0) - i_0 + 1$. Also, $n_0 \in \mathbb{N}^+$ by the precondition. So the invariant is satisfied before the first iteration.

Assume the invariant holds after the jth iteration, and assume it runs for a $j+1$th.

$n_{j+1} = n_j \mathbin{//} 2$, which, by lemma 0.1, has one less bit than n_j, since $n_j > 1$ by the loop condition. $i_{j+1} = i_j + 1$. So 2a is preserved, since we subtract 1 from each side of the equation.

Furthermore, since $n_j > 1$ by the loop condition, $n_{j+1} = n_j \mathbin{//} 2 \in \mathbb{N}^+$, as required by 2b. □

Corollary 0.2.1 (Partial correctness). *Given a valid input, if* `bitcount` *terminates, it satisfies the postcondition.*

Proof. Assume the while loop exits after some iteration j. By the loop condition, $n_j \leq 1$, and by 2b $n_j \in \mathbb{N}^+$, thus $n_j = 1$.

From 2a, it follows that

$$\begin{aligned} \mathrm{b}(n_j) &= \mathrm{b}(n_0) - i_j + 1 \\ \mathrm{b}(1) &= \mathrm{b}(n_0) - i_j + 1 \\ 1 &= \mathrm{b}(n_0) - i_j + 1 \\ \mathrm{b}(n_0) &= i_j \end{aligned}$$

So returning i_j satisfies the postcondition. □