CSC236 Homework Assignment #2

Induction Proofs on Program Correctness and Recurrences

Alexander He Meng

Prepared for October 28, 2024

Question #1

Consider the following program from pg. 53-54 of the course textbook:

```python
def avg(A):
    """
    Pre: A is a non-empty list
    Post: Returns the average of the numbers in A
    """
    sum = 0
    i = 0
    while i < len(A):
        sum += A[i]
        i += 1
    return sum / len(A)

print(avg([1, 2, 3, 4]))  # Example usage
```

Denote the predicate:

$$Q(j) : \text{At the beginning of the } j^{\text{th}} \text{ iteration, } \texttt{sum}_j = \sum_{k=0}^{i_j - 1} A[k].$$

<u>**Claim:**</u>
$\forall j \in \{1, \ldots, len(A)\}, Q(j)$

Proof.
This proof leverages the Principle of Simple Induction.

<u>Base Case:</u>
Let $j = 1$.
At the beginning of the 1^{st} iteration, $\texttt{sum}_1 = 0$ and $i_1 = 0$.

It follows that

$$\texttt{sum}_1 = \sum_{k=0}^{i_1 - 1} A[k] = \sum_{k=0}^{0-1} A[k] = \sum_{k=0}^{-1} A[k] = 0.$$

Hence, $Q(1)$.

Induction Hypothesis:
Assume for some iteration $m \in \{1, ..., len(A) - 1\}$, $Q(m)$.

Namely, for the m^{th} iteration,

$$\texttt{sum}_m = \sum_{k=0}^{i_m - 1} A[k].$$

Induction Step:
Proceed to show $Q(m+1)$:

Notice that $\texttt{sum}_{m+1} = \texttt{sum}_m + A[i_{m+1}]$, by *Line 9* of the program.

By the Induction Hypothesis,

$$\texttt{sum}_m + A[i_{m+1}] = \sum_{k=0}^{i_m - 1} A[k] + A[i_{m+1}],$$

and by *Line 10* of the program, $i_{m+1} = i_m + 1$;

$$\sum_{k=0}^{i_m - 1} A[k] + A[i_{m+1}] = \sum_{k=0}^{i_{m+1} - 1} A[k].$$

Thus,

$$\texttt{sum}_{m+1} = \sum_{k=0}^{i_{m+1} - 1} A[k]$$

as needed.

Therefore, by the Principle of Simple Induction, $Q(j)$ holds for all $j \in \{1, \ldots, len(A)\}$.

$\square$

Question #2

Recall $Q(j)$ from *Question # 1*:

$$Q(j) : \text{At the beginning of the } j^{\text{th}} \text{ iteration, } \texttt{sum}_j = \sum_{k=0}^{i_j - 1} A[k].$$

Denote the following predicate:

$$Q'(n) : 0 \leq n < len(A) \implies Q(n+1)$$

Claim:
Proving $\forall j \in \{1, \ldots, len(A)\}, Q(j)$ is equivalent to proving $\forall n \in \mathbb{N}, Q'(n)$.

Proof.
Remarks
It is sufficient to show that $\forall j \in \{1, \ldots, len(A)\}, Q(j) \iff \forall n \in \mathbb{N}, Q'(n)$, to show that proving one of these statements is equivalent to proving the other.

$(\forall j \in \{1, \ldots, len(A)\}, Q(j)) \implies (\forall n \in \mathbb{N}, Q'(n))$:
Suppose $\forall j \in \{1, \ldots, len(A)\}, Q(j)$.

Then, fix $n \in \mathbb{N}$ and suppose $0 \leq n < len(A)$.

Because $n \in \{0, ..., len(A) - 1\}$, it follows that $(n+1) \in \{1, \ldots, len(A)\}$.

By assumption, $Q(n+1)$.

Thus, $\forall n \in \mathbb{N}, Q'(n)$.

$(\forall j \in \{1, \ldots, len(A)\}, Q(j)) \impliedby (\forall n \in \mathbb{N}, Q'(n))$:
Suppose $\forall n \in \mathbb{N}, Q'(n)$.

Let $j \in \{1, \ldots, len(A)\}$.

Then, $(j-1) \in \mathbb{N}$.

It follows that $0 \leq j-1 \leq len(A)-1$.

Since $len(A)-1 < len(A)$, $0 \leq j-1 < len(A)$.

By assumption, $Q((j-1)+1)$.

Thus, $\forall j \in \{1, \ldots, len(A)\}, Q(j)$.

Conclusion:
Therefore, $\forall j \in \{1, \ldots, len(A)\}, Q(j) \iff \forall j \in \mathbb{N}, Q'(n)$.

$\square$

Question #3

As follows below, Q6-Q10 respectively represent questions 6 through 10 from pp. 64-66 of the course textbook.

Q6:

Consider the following code:

```
1 def f(x):
2     """Pre: x is a natural number"""
3     a = x
4     y = 10
5     while a > 0:
6         a -= y
7         y -= 1
8     return a * y
```

(a): Loop Invariant Which Characterizes `a` and `y`:

For arbitrary natural n...

Let $i_1 = 0$ and $i_n = i_{n-1} + 1$.

Let y_n be the value of y before the $(n+1)$th iteration. By *Line 4* (initializes $y = 10$) and *Line 7* (decrements y by 1) of the program, $y_n = 10 - \sum_{q=1}^{n} 1 = 10 - n \times 1 = 10 - n$.

Denote the loop invariant:

$$P(j) : (a_j = x - \sum_{k=0}^{i_j - 1} y_k) \land (y_j = 10 - j)$$

For example, before the 1st iteration, $a_1 = x - \sum_{k=0}^{i_1-1} y_k = x - \sum_{k=0}^{0-1} y_k = x - 0 = x$.

Before the 2nd iteration, $a_2 = x - \sum_{k=0}^{i_2-1} y_k = x - \sum_{k=0}^{1-1} y_k = x - y_0 = x - 10$.

(b): Why This Function Fails to Terminate

Suppose $x > \sum_{k=1}^{10} k = 55$.

By $P(j)$, before the 11th iteration, $a_{11} = x - \sum_{k=0}^{i_{11}-1} y_k = x - \sum_{k=0}^{10-1} y_k = x - \sum_{k=0}^{9} (10 - k) =$

$x - [10\sum_{k=0}^{9}(1) - \sum_{k=0}^{9}(k)] = x - [10(10) - \frac{9(9+1)}{2}] = x - [100 - 45] = x - 55.$

Since $x > 55$, it follows that $a_{11} = x - 55 > 0$.

As well, y_{10} (the value of y after the 11^{th} iteration) is $10 - 11 = -1$.

Notice that in all subsequent iterations, a will decrement by $y_n < 0|_{n\geq 11}$ (where n is the iteration number of the corresponding iteration).

Since a decrements by a negative number subsequently, the loop causes a to grow large, thereby retaining $a > 0$.

Thus, the function fails to terminate for $x > 55$ (because $\neg(a > 0)$ is never satisfied).

Q7:

(a) Consider the recursive program below:

```
def exp_rec(a, b):
    if b == 0:
        return 1
    else if b mod 2 == 0:
        x = exp_rec(a, b / 2)
        return x * x
    else:
        x = exp_rec(a, (b - 1) / 2)
        return x * x * a
```

Preconditions:

$$(b \in \mathbb{N}) \land (a \neq 0)$$

Postconditions:

$$\text{Returns } a^b.$$

Denote the following predicate:

$$P(b) : \text{The program returns } a^b.$$

<u>Claim:</u> $\forall b \in \mathbb{N}, P(b)$

Proof.
This proof explores the Principle of Complete Induction on b.

Fix $a \neq 0$.

<u>Base Case:</u>
Let $b = 0$.

Then, by *Lines 2-3* of the program, the program returns $1 = a^0 = a^b$.

Hence, $P(0)$.

<u>Induction Hypothesis:</u>
Assume for some $k \in \mathbb{N}$ and for all $l \in [0, k] \cap \mathbb{N}$, $P(l)$.

This means the program returns a^l for every l as described.

<u>Induction Step:</u>
Proceed to show $P(k + 1)$ with case analysis:

Case 1 - Suppose $(k + 1)(mod\ 2) \not\equiv 0$*:*
Then, program again enters the `else` statement in *Line 7*.

Here, the program sets x to `exp_rec(a, ((k + 1) - 1) / 2)`.

Notice that `exp_rec(a, ((k + 1) - 1) / 2)` $=$ `exp_rec(a, (k / 2))`.

Since $(k+1)(\text{mod } 2) \not\equiv 0$, it must be that $k(\text{mod } 2 \equiv 0)$.

Thus, $\frac{k}{2} \in \mathbb{N}$ and $\frac{k}{2} < k$.

By the Induction Hypothesis, `exp_rec(a, k / 2)` returns $a^{\frac{k}{2}}$.

Finally, the original function call returns $x \times x \times a$, which evaluates to $a^{\frac{k}{2}} \times a^{\frac{k}{2}} \times a = a^{\frac{k}{2}+\frac{k}{2}+1} = a^{k+1}$, as needed.

Thus, $P(k+1)$ holds.

Case 2 - Suppose $(k+1)(mod\ 2) \equiv 0$*:*
Then, the program reaches *Line 5* and sets x to `exp_rec(a, (k + 1) / 2)`.

Notice that $\frac{k+1}{2} \in \mathbb{N}$ and $\frac{k+1}{2} \leq k$.

By the Induction Hypothesis, `exp_rec(a, (k + 1) / 2)` returns $a^{\frac{k+1}{2}}$.

Finally, the original function call returns $x \times x$, evaluating to $a^{\frac{k+1}{2}} \times a^{\frac{k+1}{2}} = a^{\frac{k+1}{2}+\frac{k+1}{2}} = a^{k+1}$, as needed.

Thus, P(k + 1) holds.

<u>Conclusion:</u>
Therefore, $P(k+1)$ holds in all cases.

By the Principle of Complete Induction, $\forall b \in \mathbb{N}, P(b)$.

$\square$

(b) Consider the iterative version of the previous program:

```
def exp_iter(a, b):
    ans = 1
```

```
    mult = a
    exp = b
    while exp > 0:
        if exp mod 2 == 1:
            ans *= mult
        mult = mult * mult
        exp = exp // 2
    return ans
```

Preconditions:

$$(b \in \mathbb{N}) \wedge (a \neq 0)$$

Postconditions:

$$\text{Returns } a^b.$$

Claim: For all natural b, the program returns a^b and terminates.

Proof.
Fix $a \neq 0$ and $b \in \mathbb{N}$.

Loop Invariant Proof:
Denote the Loop Invariant:

$$P(i) : a^b = \texttt{mult}_i^{\texttt{exp}_i} \times \texttt{ans}_i$$

To prove the loop invariant, this proof explores the Principle of Simple Induction on $i \in [1, \lfloor log_2 b \rfloor + 2] \cap \mathbb{N}$.

Base Case:
Let $i = 1$.

Then, the program retains the values $\texttt{mult}_1 = a$, $\texttt{exp}_1 = b$, `ans_1 = 1`.

Notice that $a^b = a^b \times 1 = \texttt{mult}_1^{\texttt{exp}_1} \times \texttt{ans}_1$.

Hence, at the beginning of the 1st iteration, $P(1)$.

<u>Induction Hypothesis:</u>
Assume for some $k \in [1, \lfloor log_2 b \rfloor + 1] \cap \mathbb{N}, P(k)$;

$$P(k) : (a^b = \texttt{mult}_k^{\texttt{exp}_k} \times \texttt{ans}_k).$$

<u>Induction Step:</u>
Notice that $\texttt{exp}_k = \lfloor \frac{b}{2^{k-1}} \rfloor$.

Because $k - 1 \leq \lfloor log_2 b \rfloor$, then $2^{k-1} \leq 2^{\lfloor log_2 b \rfloor} \leq 2^{log_2 b} = b$.

Therefore, it follows that $\frac{b}{2^{k-1}} \geq 1$.

Thus, $\texttt{exp}_k = \lfloor \frac{b}{2^{k-1}} \rfloor \geq 1$, and the following iteration runs.

Then, the program yields the following values:

$$\begin{cases} \texttt{mult}_{k+1} = \texttt{mult}_k \times \texttt{mult}_k = \texttt{mult}_k^2, & \text{by } \textit{Line 8} \\ \texttt{exp}_{k+1} = \left\lfloor \frac{\texttt{exp}_k}{2} \right\rfloor, & \text{by } \textit{Line 9} \end{cases}$$

Notice that $\texttt{mult}_{k+1}^{\texttt{exp}_{k+1}} = (\texttt{mult}_k^2)^{\lfloor \frac{\texttt{exp}_k}{2} \rfloor} = \texttt{mult}_k^{2\lfloor \frac{\texttt{exp}_k}{2} \rfloor}$.

Proceed to show $P(k+1)$ with case analysis:

<u>*Case 1 - Suppose* $exp_k (mod\ 2) \equiv 1$*:*</u>
By *Lines 6-7* of the program, $\texttt{ans}_{k+1} = \texttt{ans}_k \times \texttt{mult}_k$.

So, $(\texttt{mult}_{k+1}^{\texttt{exp}_{k+1}}) \times (\texttt{ans}_{k+1}) = (\texttt{mult}_k^{2\lfloor \frac{\texttt{exp}_k}{2} \rfloor}) \times (\texttt{ans}_k \times \texttt{mult}_k)$.

Since $\texttt{exp}_k (\text{mod } 2) \equiv 1$, it follows that $2\lfloor \frac{\texttt{exp}_k}{2} \rfloor = \cancel{2}(\frac{\texttt{exp}_k - 1}{\cancel{2}}) = \texttt{exp}_k - 1$.

Thus,

$$\begin{aligned}(\texttt{mult}_k^{2\lfloor\frac{\texttt{exp}_k}{2}\rfloor}) \times (\texttt{ans}_k \times \texttt{mult}_k) &= (\texttt{mult}_k^{\texttt{exp}_k - 1}) \times (\texttt{mult}_k \times \texttt{ans}_k) \\ &= \texttt{mult}_k^{\texttt{exp}_k - 1} \times \texttt{mult}_k \times \texttt{ans}_k \\ &= (\texttt{mult}_k^{\texttt{exp}_k - 1} \times \texttt{mult}_k) \times \texttt{ans}_k \\ &= (\texttt{mult}_k^{(\texttt{exp}_k - 1)+1}) \times \texttt{ans}_k \\ &= \texttt{mult}_k^{\texttt{exp}_k} \times \texttt{ans}_k \\ &= a^b,\end{aligned}$$

by the Induction Hypothesis.
Therefore, $(\texttt{mult}_{k+1}^{\texttt{exp}_{k+1}}) \times (\texttt{ans}_{k+1}) = a^b$; $P(k+1)$ holds.

Case 2 - Suppose $exp_k (mod\ 2) \not\equiv 1$:
By *Line 6* of the program, *Line 7* does not run.

Hence, $\texttt{ans}_{k+1}$ retains the value as represented by $\texttt{ans}_k$; $\texttt{ans}_{k+1} = \texttt{ans}_k$.

So, $(\texttt{mult}_{k+1}^{\texttt{exp}_{k+1}}) \times (\texttt{ans}_{k+1}) = (\texttt{mult}_k^{2\lfloor\frac{\texttt{exp}_k}{2}\rfloor}) \times \texttt{ans}_k$.

Notice that $\texttt{exp}_k (\text{mod } 2) \not\equiv 1 \iff \texttt{exp}_k (\text{mod } 2) \equiv 0$.

So, $2\lfloor\frac{\texttt{exp}_k}{2}\rfloor = \cancel{2}(\frac{\texttt{exp}_k}{\cancel{2}}) = \texttt{exp}_k$.

Then, it follows that $(\texttt{mult}_k^{2\lfloor\frac{\texttt{exp}_k}{2}\rfloor}) \times \texttt{ans}_k = (\texttt{mult}_k^{\texttt{exp}_k}) \times \texttt{ans}_k = a^b$, by the Induction Hypothesis.

Still, $(\texttt{mult}_{k+1}^{\texttt{exp}_{k+1}}) \times (\texttt{ans}_{k+1}) = a^b$; $P(k+1)$ likewise holds.

Conclusion of Loop Invariant Proof:
Collectively, $P(k+1)$ holds in all cases.

By the Principle of Simple Induction, $P(i)$ holds for all $i \in [1, \lfloor log_2 b\rfloor + 2]$.

<u>Program Termination Proof</u>

Notice that *Line 9* of the program performs floor division by 2 on `exp` in each iteration.

From continual division, `exp` eventually becomes small enough that it reaches 0 through the next floor division by 2.

Since the program's loop requires `exp` > 0 to run, having `exp` reach 0 indeed terminates the loop.

<u>Conclusion:</u>

Therefore, this program is both correct (by the loop invariant) and terminates.

□

Q8

Consider the following linear time program:

```python
def majority(A):
    """
    Pre: A is a list with more than half its entries equal to x
    Post: Returns the majority element x
    """
    c = 1
    m = A[0]
    i = 1
    while i <= len(a) - 1:
        if c == 0:
            m = A[i]
            c = 1
        else if A[i] == m:
            c += 1
        else:
            c -= 1
        i += 1
    return m
```

Claim: For all lists A with more than half its entries equal to x, the program returns the majority element x and terminates.

Proof.
For simplicity, express "`List` has more than half its entries equal to x" by "`List` is valid," and its complement by "`List` is NOT valid."

Let v_n represent the difference between the count of x and the count of elements that are not x in sublist $A[0:n]$, before the n^{th} iteration.

Loop Invariant Proof:
Denote the Loop Invariant:

$$P(i):$$

$$((A[0:i] \text{ is valid}) \implies ((m_i = x) \wedge (c_i \geq v_i)))$$

$$\wedge$$

$$((A[0:i] \text{ is NOT valid}) \implies ((m_i = x) \vee (c_i \leq -v_i)))$$

To prove the loop invariant, this proof explores the Principle of Simple Induction on $i \in [1, len(a)]$.

Base Case:
Let $i = 1$.

Then, $A[0:i] = A[0:1] = [A[0]] = [x]$, by the precondition.

With the sole element being x, $A[0:i]$ is valid.

Notice that *Line 7* of the program sets m_1 to $A[0] = x$, and $c_1 = 1 = v_1$.

Thus, $m_1 = x$ and $c_1 \geq v_1$; $P(1)$ holds.

Induction Hypothesis:

Assume for some $k \in [1, len(a) - 1]$, $P(k)$:

$$P(k):$$

$$((A[0:k] \text{ is valid}) \implies ((m_k = x) \land (c_k \geq v_k)))$$

$$\land$$

$$((A[0:k] \text{ is NOT valid}) \implies ((m_k = x) \lor (c_k \leq -v_k)))$$

Induction Step:
Consider the following cases...

Suppose $A[0:k+1]$ is valid:
By the Induction Hypothesis, $m_k = x$ and $c_k \geq v_k$.

Notice that $v_k > 0$ because $A[0:k+1]$ is valid (the sublist has more entries of x than entires of not x).

It follows that $c_k \geq v_k > 0$, so $c \neq 0$.

When the iteration runs, the program does not enter *Lines 10-12.*

So, m_{k+1} retains the value of $m_k = x$, and $CONTINUEHERE$!!!.

Suppose $A[0:k+1]$ is NOT valid:

Conclusion of Loop Invariant Proof

wordsgohere

Program Termination Proof:

wordsgohere

Conclusion:

wordsgohere

□

Q9

Consider the bubblesort algorithm as follows:

```
def bubblesort(L):
    """
    Pre: L is a list of numbers
    Post: L is sorted
    """
    k = 0
    while k < len(L):
        i = 0
        while i < len(L) - k - 1:
            if L[i] > L[i + 1]:
                swap L[i] and L[i + 1]
            i += 1
        k += 1
```

(a): Denote the inner loop's invariant:

$$P(j) : (\forall i \in [0, j-1] \cap \mathbb{N})(L[i] \leq L[j])$$

Claim: At the start of all iterations $j \in [1, len(L) - k] \cap \mathbb{N}$, $P(j)$.

Proof.

Base Case:

Let $j = 1$. Then, $i \in [0, 1-1] \cap \mathbb{N}$.

Then, $i = 0$.

Notice that $L[0] \leq L[1]$, by *Lines 10-11* of the program (if this is not satisfied, the elements are swapped so that it is).

Thus, $P(1)$.

<u>Induction Hypothesis:</u>
Assume for some $k \in [1, len(L) - k - 1] \cap \mathbb{N}$, $P(k)$ holds.

This means, $(\forall i \in [0, k-1] \cap \mathbb{N})(L[i] \leq L[k])$.

<u>Induction Step:</u>
Suppose $L[j] \leq L[j+1]$.

Then, no swaps are made and $P(j+1)$ immediately holds by the Induction Hypothesis.

So, consider $L[j] > L[j+1]$.

By *Lines 10-11* of the program, the two elements are swapped.

The result is $L[j] \leq L[j+1]$ before the $(j+1)^{\text{th}}$ iteration, and $P(j+1)$ likewise holds by the Induction Hypothesis.

$\square$

(b): Denote the outer loop's invariant:

$$Q(n) : L[len(L) - n :] \text{ is sorted.}$$

<u>Claim:</u> At the start of all iterations $n \in [1, len(L)]$, $Q(n)$.

Proof.

Base Case:

Let $n = 1$.

Then, $L[len(L) - n :] = L[len(L) - 1 :]$ is a sublist of L containing only the last element of L.

Vacuously, this list is indeed sorted, so $Q(1)$.

Induction Hypothesis:

Assume for some $m \in [1, len(L)]$, $Q(m)$.

This means $L[len(L) - m :]$ is sorted.

Induction Step:

By the inner loop of the program $(\forall i \in [0, m - 1] \cap \mathbb{N})(L[i] \leq L[m])$.

This means, there is no element larger than $L[len(L) - m]$ for elements in indices less than $len(L) - m$.

The inner loop places this large value at $L[len(L) - m]$.

By the Induction Hypothesis, $L[len(L) - m :]$ is sorted.

In the subsequent iteration, the program's inner loop grabs a new element from the pool of elements not larger than $L[len(L) - m]$ (from the smaller indices).

This inner loop places this new large value at $L[len(L) - (m + 1)]$.

Since this $L[len(L) - (m + 1)]$ is not larger than $L[len(L) - m]$, and because $L[len(L) - m :]$ is sorted, $L[len(L) - (m + 1) :]$ is updated as a sorted sublist.

Thus, $Q(m+1)$.

$\square$

(c): <u>Claim:</u> If L is a list of numbers, then the program returns L as a sorted list.

Proof.
To show that this program is correct, it remains to show that the inner and outer loops both terminate, since their invariants are proven.

<u>Inner Loop Termination:</u>
To show that the inner loop terminates, consider the loop variant $Var = len(L) - k - i$.

Denote $\widetilde{Var}$ as the loop variant in the subsequent iteration.

Then, notice that $\widetilde{Var} = len(L) - k - (i+1) < len(L) - k - i = Var$.

Since $len(L), k, i \in \mathbb{N}$, and the variant decreases in subsequent iterations, the inner loop indeed terminates.

<u>Outer Loop Termination:</u>
To show that the outer loop terminates, consider the loop invariant $Var = len(L) - i$.

Denote $\widetilde{Var}$ as the loop variant in the subsequent iteration.

Then, notice that $\widetilde{Var} = len(L) - (i+1) < len(L) - i = Var$.

Likewise, since $len(L), i \in \mathbb{N}$, and the invariant decreases in subsequent iterations, the outer loopp terminates as well.

<u>Conclusion:</u>
Therefore, because both the inner and outer loop are correct and terminate, the program

correctly takes any list of numbers L and returns its corresponding sorted list.

□

Q10

Consider the following generalization of the `min` function:

```
def extract(A, k):
    pivot = A[0]
    # Use partition from quicksort
    L, G = partition(A[1, ..., len(A) - 1], pivot)
    if len(L) == k - 1:
        return pivot
    else if len(L) >= k:
        return extract(L, k)
    else:
        return extract(G, k - len(L) - 1)
```

(a): Proof of Correctness

<u>Preconditions:</u>

$$A \text{ is a list of numbers such that } len(A) \geq 1, \text{ and } k \in [1, len(A)] \cap \mathbb{N}.$$

<u>Postconditions:</u>

$$\text{Returns the } k^{\text{th}} \text{ smallest element of } A.$$

Denote the following predicate:

$$P(n) : \text{The program returns the } 1 \leq k^{\text{th}} \leq n \text{ smallest element from the list } A \text{ of size } n.$$

<u>Claim:</u> For any list A of length $n = len(A) \geq 1$, $P(n)$ holds.

Proof.
This proof explores the Principle of Complete Induction on n.

Fix $k \in [1, n] \cap \mathbb{N}$.

Base Case:
Let A be a list of length $n = len(A) = 1$.
Then, A is a list containing one element, and $k = 1$ (representing the *first smallest* element of A) is the only value of k.

By *Line 2* of the program, `pivot` $= A[0]$.

Because `pivot` is an element of A, the precondition for the function $partition(A, \texttt{pivot})$ is satisfied.

So, assume the corresponding postcondition of $partition(A, \texttt{pivot})$:

L contains the elements of A less than the pivot,
and G contains the elements greater than the pivot.

With no other elements to compare with, both L and G will be empty.

In *Line 5* of the program, notice that $len(L) == k - 1$ holds as $len(L) = 0 = (1) - 1$.

Therefore, the program enters *Line 6* and returns `pivot`.

As the sole and smallest element of A, returning `pivot` indeed returns the 1^{st} smallest element of A.

Thus, $P(1)$.

Induction Hypothesis:
Assume for all lists A of length $j \in [1, i] \cap \mathbb{N}$, for some $i \geq 1$ and $k \in [1, j] \cap \mathbb{N}$, $P(j)$ holds.

This means all lists $A_1, A_2, \ldots, A_i$ passed into $extract(A, k)$ result in $extract(A, k)$ returning the k_1^{th} or k_2^{th} or $\ldots$ or k_i^{th} corresponding smallest element from the corresponding list.

Induction Step:

Assume the algorithm is correct for all lists of length $j \in [1, i] \cap \mathbb{N}$ for some $i \geq 1$ and $k \in [1, j] \cap \mathbb{N}$.

To show $P(1), P(2), \ldots, P(i) \implies P(i+1)$, it is necessary to show that the function algorithm is correct for all lists of size $i + 1$ and $k \in [1, i + 1] \cap \mathbb{N}$ for some $i \geq 1$ and $k \in [1, j] \cap \mathbb{N}$, while assuming the preconditions hold on the function call.

Notice that the program sets `pivot` to $A[0]$, the first element in A.

So, `pivot` represents an element of A, satisfying the precondition for the *partition* function.

partition then returns L and G, which are lists containing all elements less than the pivot and greater than the pivot, respectively.

Consider the following cases...

Suppose $len(L) = k - 1$:
Then, there are $k - 1$ elements smaller than the pivot, by the postcondition of *partition*.

Therefore, `pivot` must be the k^{th} smallest element in A.

The program enters *Line 6* and returns `pivot`.

Thus, $P(i + 1)$.

Suppose $len(L) \geq k$:
Then, L has enough elements to contain the k^{th} smallest element in A, so the k^{th} smallest element in A must be in L.

Notice, additionally, that `pivot` is not the k^{th} smallest element.

Then, the program reaches *Line 8*, returning $extract(L, k)$.

Notice that L is a list with $len(L) \geq 1$, and $k \in [1, len(L)] \cap \mathbb{N}$.

Then, by the Induction Hypothesis (as $len(L) < len(A)$), $extract(L, k)$ correctly returns the k^{th} smallest element in the list L.

Thus, $P(i + 1)$.

Suppose $len(L) < k - 1$:
Then, L does not have enough elements to contain the k^{th} smallest element in A, so the k^{th} smallest element in A must be G.

Again, `pivot` is not the k^{th} smallest element.

Then, the program reaches *Line 10*, returning $extract(G, k - len(L) - 1)$.

Notice that the $(k - len(L) - 1)^{\text{th}}$ smallest element in G is the smallest element in A (exclude the number of elements in L and `pivot`).

Notice that G is a list with $len(G) \geq 1$, and $(k - len(L) - 1) \in [1, len(G)] \cap \mathbb{N}$.

Then, by the Induction Hypothesis (as $len(G) < len(A)$), $extract(G, k - len(L) - 1)$ correctly returns the $(k - len(L) - 1)^{\text{th}}$ smallest element in the list G.

Thus, $P(i + 1)$.

Conclusion:
Since the Base Case $P(1)$ holds, and $P(1), P(2), \ldots, P(i) \implies P(i + 1)$, by the Principle of Complete Induction, $P(n)$ holds for all lists of length $n \geq 1$.

Termination Proof:

The above proof shows that each call to $extract(A, k)$ considers a sublist with a length less than the initial list passed to the function, from either a previous call up the stack or the initial call.

In each recursive call case, either L or G is the sublist being passed.

Notice that, excluding `pivot`, each of these lists are at most as long as A.

Therefore, $len(G) \leq len(A) - 1$ and $len(L) \leq len(A) - 1$.

As a note, the equalities hold when `pivot` is the smallest element in A and when `pivot` is the greatest element in A, respectively.

Citation:
This proof is inspired by the Tutorial 5 slides for its Problem 1 from this CSC236 course.

$\square$

(b): Worst-Case Runtime
Denote the size of the list A by $n = len(A)$.

To find the worst-case runtime of the program, consider the program's behaviour for its base and recursive cases.

Base Case of Program:
Let n = 1.

Then, the program reaches the *if* block (for the reason as analysed in the above proof) and returns `pivot`.

Exiting here, the program runs in constant time.

Base Case of Program:

Let $n > 1$.

In quicksort partitioning, the worst case occurs when one of the lists L, G is empty.

Then, the $elseif$ and $else$ blocks run, making a function call on a sublist that is 1 size smaller than the size of the current list.

Also, the number of recursive steps depend on the size of the current list, n.

This means, the recursive case has runtime $T(n-1) + dn$.

Define the recurrence:

$$T(n) = \begin{cases} c & \text{if } n \leq 1 \\ T(n-1) + dn & \text{if } n > 1 \end{cases}$$

For $n \leq 1$, $T(n) = c$, for some constant c, and the program runs in constant time with the tight bound $\Theta(1)$.

Otherwise, $n > 1$; evaluate the closed form for $T(n)$ as follows:

$$\begin{aligned} T(n) &= T(n-1) + dn \\ &= T((n-1)-1) + [dn] \\ &= T((n-2)-1) + [dn] \\ &. \dots \\ &. \dots \\ &. \dots \\ &= c + d(\sum_{i=0}^{k}(n-1)) \end{aligned}$$

Through repeated substitution, $T(0)$ is eventually reached.

Notice that every unique call of $T(n)$ has the form $T((n-k)-1)$, for some natural k.

Therefore, $T(0) = T((n-k)-1) \implies 0 = (n-k)-1 \implies k = n-1$; the call of $T(0)$ occurs when $k = n-1$.

Continue evaluating $T(n)$ with $k = n-1$:

$$\begin{aligned} T(n) &= c + d(\sum_{i=0}^{n-1}(n-1)) \\ &= d[(\sum_{i=0}^{n-1} n) - (n-1)] + c \\ &= d[(\frac{((n-1)+1)(n-1)}{2}) - (n-1)] + c \\ &= d[(\frac{n^2 - n - 2n + 2}{2})] + c \\ &= \frac{dn^2 - 3dn + 2c}{2} \end{aligned}$$

Thus, $T(n) = \frac{dn^2-3dn+2c}{2}$ for $n > 1$.

Notice that in this closed form, n^2 is the term with the highest degree, so it takes precedence.

Therefore, the tight asymptotic bound on the runtime of the program is $\Theta(n^2)$.

By definition, the worst-case runtime of the program is $\mathcal{O}(n^2)$.

<u>Final Answer:</u>
The worst-case runtime of the program is $\mathcal{O}(n^2)$.

Question #4

As follows below, VI, VII, XII, and XIV respectively represent questions 6, 7, 12, and 14 from pp. 46-48 of the course textbook.

VI
Let $T(n)$ be the number of binary strings of length n where every 1 is immediately preceded by a 0.

(a): Recurrence for $T(n)$:

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

(b): Closed Form Expression for $T(n)$:

Notice that $T(n)$ is equivalent to the Fibonacci sequence, shifted left by one term.

Therefore, $T(n) = F_{n+1}$, where F_n is the n^{th} value of the Fibonacci sequence.

Then, the closed form for $T(n)$ is as follows:

$$T(n) = \frac{\left(\frac{1+\sqrt{5}}{2}\right)^{n+1} - \left(\frac{1-\sqrt{5}}{2}\right)^{n+1}}{\sqrt{5}}$$

(c): Proof of Correctness of Closed Form Expression
Denote the following predicate:

$$P(n) : T(n) \text{ represents the number of binary strings of length } n$$

where every 1 is immediately preceded by a 0.

<u>Claim:</u> $\forall n \in \mathbb{N}, T(n)$

Proof.

<u>Base Cases:</u>

Let $n = 0$.

Then,

$$T(0) = \frac{\left(\frac{1+\sqrt{5}}{2}\right)^{\cancelto{1}{0+1}} - \left(\frac{1-\sqrt{5}}{2}\right)^{\cancelto{1}{0+1}}}{\sqrt{5}} = \frac{\frac{1+\sqrt{5}}{2} - \frac{1-\sqrt{5}}{2}}{\sqrt{5}} = \frac{\frac{\cancel{2}\sqrt{5}}{\cancel{2}}}{\sqrt{5}} = \frac{\cancel{\sqrt{5}}}{\cancel{\sqrt{5}}} = 1.$$

Indeed, there is only one binary string (the empty string) with length 0, there are simply no 1's.

Thus, "0 immediately precedes every 1" is vacuously true; $T(0)$.

Let $n = 1$.

Then,

$$T(1) = \frac{\left(\frac{1+\sqrt{5}}{2}\right)^{\cancelto{2}{1+1}} - \left(\frac{1-\sqrt{5}}{2}\right)^{\cancelto{2}{1+1}}}{\sqrt{5}} = \frac{\frac{1+2\sqrt{5}+5}{4} - \frac{1-2\sqrt{5}+5}{4}}{\sqrt{5}} = \frac{\frac{\cancel{4}\sqrt{5}}{\cancel{4}}}{\sqrt{5}} = \frac{\cancel{\sqrt{5}}}{\cancel{\sqrt{5}}} = 1.$$

If the string solely contains 0, then there is simply no 1 in the string; vacuously, this is a valid string.

If the string solely contains 1, then it is clearly not preceded by a 0 which does not fit (string has length 1).

Therefore, there is only 1 valid string of length 1; $P(1)$ holds.

Induction Hypothesis:

Assume for some $k \in \mathbb{N}$, for all $m \in [0, k], P(m)$.

This means for some natural k, for all strings of length $m \in [0, k]$, there are $T(m)$ binary strings where every 1 is immediately preceded by a 0.

Induction Step:

Evaluate $T(m + 1)$ as follows:

$$
\begin{aligned}
T(m+1) &= T((m+1)-1) + T((m+1)-2) \\
&= T(m) + T(m-1) \\
&= \frac{(\frac{1+\sqrt{5}}{2})^{m+1} - (\frac{1-\sqrt{5}}{2})^{m+1}}{\sqrt{5}} + \frac{(\frac{1+\sqrt{5}}{2})^{\overset{m}{\cancel{(m-1)+1}}} - (\frac{1-\sqrt{5}}{2})^{\overset{m}{\cancel{(m-1)+1}}}}{\sqrt{5}} \\
&= \frac{(\frac{1+\sqrt{5}}{2})^{m+1} - (\frac{1-\sqrt{5}}{2})^{m+1} + (\frac{1+\sqrt{5}}{2})^{m} - (\frac{1-\sqrt{5}}{2})^{m}}{\sqrt{5}} \\
&= \frac{(\frac{1+\sqrt{5}}{2})^{m+1}[1 + \frac{2}{1+\sqrt{5}}] - (\frac{1-\sqrt{5}}{2})^{m+1}[1 + \frac{2}{1-\sqrt{5}}]}{\sqrt{5}} \\
&= \frac{(\frac{1+\sqrt{5}}{2})^{m+1}(\frac{1+\sqrt{5}}{2}) - (\frac{1-\sqrt{5}}{2})^{m+1}(\frac{1-\sqrt{5}}{2})}{\sqrt{5}} \\
&= \frac{(\frac{1+\sqrt{5}}{2})^{(m+1)+1} - (\frac{1-\sqrt{5}}{2})^{(m+1)+1}}{\sqrt{5}}
\end{aligned}
$$

Note that $1 + \frac{2}{1+\sqrt{5}} = \frac{1+\sqrt{5}}{2}$ and $1 + \frac{2}{1-\sqrt{5}} = \frac{1-\sqrt{5}}{2}$.

Therefore, $T(m+1)$ holds for the recurrence.

Conclusion:
By the Principle of Complete Induction, the $T(n)$ holds for all $n \in \mathbb{N}$.

$\square$

VII

Let $T(n)$ denote the number of distinct full binary trees with n nodes. For example, $T(1) = 1$, $T(3) = 1$, and $T(7) = 5$. Note that every full binary tree has an odd number of nodes.

Recurrence for $T(n)$:

The recursive part of the recurrence of $T(n)$ connects the number of nodes to the unique number of full binary trees.

As well, the left and right subtrees can at most have $\frac{n-3}{2}$ nodes.

Finally, when the left subtree has $2k+1$ nodes and the right subtree has $n-1-(2k+1)$ nodes, there are $T(2k+1)T(n-1-(2k+1))$ full binary trees.

Altogether, for particular numbers of nodes for the left and right subtree, the sum of all of these combinations is obtained.

Therefore, denote the recurrence as follows:

$$T(n) = \begin{cases} 1 & \text{if } n = 1 \\ \sum_{k=0}^{\frac{n-3}{2}} T(2k+1)T(n-1-(2k+1)) & \text{if } n > 1 \\ 0 & \text{if } n \text{ is even} \end{cases}$$

Claim: $T(n) \geq (\frac{1}{n})(2)^{(n-1)/2}$

Proof.

Base Case:

Let $n = 1$.

Then, $T(1) = 1$ by the recurrence.

Notice that $(\frac{1}{1})(2)^{(1-1)/2} = 2^0 = 1$.

Indeed, $T(1) = 1 \leq 1$; $T(1)$.

Induction Hypothesis:

Assume for some odd $k \in \mathbb{N}$, for all odd $m \in [1, k] \cap \mathbb{N}, T(m)$.

This means, for some odd natural number k, the number of distinct full binary trees with m nodes is less than or equal to $(\frac{1}{n})(2)^{(n-1)/2}$, for all smaller natural numbers $m \in [1, k] \cap \mathbb{N}$

Induction Step:

Consider $T(m + 2)$.

By the recurrence definition,

$$
\begin{aligned}
T(m+2) &= \sum_{k=0}^{\frac{(m+2)-3}{2}} T(2k+1)T(n-1-(2k+1)) \\
&= \sum_{k=0}^{\frac{m-1}{2}} T(2k+1)T(n-1-(2k+1)).
\end{aligned}
$$

By the Induction Hypothesis,

$$
\begin{aligned}
&\sum_{k=0}^{\frac{m-1}{2}} T(2k+1)T(n-1-(2k+1)) \\
\geq &\sum_{k=0}^{\frac{m-1}{2}} ((\frac{1}{2k+1})(2)^{((2k+1)-1)/2})((\frac{1}{n-1-(2k+1)})(2)^{((n-1-(2k+1))-1)/2})
\end{aligned}
$$

Therefore, by the Principle of Complete Induction, the claim holds.

$\square$

XII

Consider the following function:

```
def fast_rec_mult(x, y):
    n = length of x  # Assume x and y have the same length
    if n == 1:
        return x * y
    else:
        a = x // 10^(n // 2)
        b = x % 10^(n // 2)
        c = y // 10^(n // 2)
        d = y % 10^(n // 2)
        p = fast_rec_mult(a + b, c + d)
```

```
11          r = fast_rec_mult(a, c)
12          u = fast_rec_mult(b, d)
13
14          return r * 10^n + (p - r + u) * 10^(n // 2) + u
```

Worst-Case Runtime Analysis:

To find the worst-case runtime of the program, consider the program's behaviour for its base and recursive cases.

Base Case of Program:

Let $n = 1$.

Then, by *Lines 3-4*, the program returns shortly, so it runs in constant time.

Recursive Case of Program:

Let $n > 1$.

Then, *Lines 6-9* run in constant time, preparing $\frac{n}{2}$ for the recursive calls in the subsequent lines.

To follow, *Lines 10-12* has runtime $3T(\frac{n}{2})$.

What remains is the function's return statement on *Line 14*, which runs in constant time.

This means, the recursive case has runtime $3T(\frac{n}{2}) + c$, for some constant c.

Define the recurrence:

$$T(n) = \begin{cases} c & \text{if } n = 1 \\ 3T(\frac{n}{2}) + c & \text{if } n > 1 \end{cases}$$

By the Master Theorem, $a = 3, b = 2, k = 0$; c is some constant.

Notice that $a > b^k \implies 3 > 2^0 \implies 3 > 1$ (which is true).

Therefore, $T(n) \in \Theta(n^{log_b a})$.

By definition, $T(n) \in \Theta(n^{log_b a}) \implies T(n) \in \mathcal{O}(n^{log_b a})$.

Because $log_b a = log_2 3$, the worst-case runtime of the program is $\mathcal{O}(n^{log_2 3})$.

Final Answer:
The worst-case runtime of the program is $\mathcal{O}(n^{log_2 3})$.

XIV
Recall the recurrence for the worst-case runtime of quicksort:

$$T(n) = \begin{cases} c & \text{if } n \leq 1 \\ T(|L|) + T(|G|) + dn & \text{if } n > 1, \end{cases}$$

where L and G are the partitions of the list to sort.

For simplicity, ignore that each list has size $\frac{n-1}{2}$.

(a): Assume the lists are always evenly split; that is, $|L| = |G| = \frac{n}{2}$ at each recursive call.

Tight Asymptotic Bound on the Runtime of Quicksort:
If $n \leq 1$, then $T(n) = c$, for some constant c, and the program runs in constant time with the tight bound $\Theta(1)$.

Otherwise, when $n > 1$, $T(n) = T(|L|) + T(|G|) + dn$.

With $|L| = |G| = \frac{n}{2}$, it follows that $T(n) = T(\frac{n}{2}) + T(\frac{n}{2}) + dn = 2T(\frac{n}{2}) + dn$.

By the Master Theorem, $a = 2, b = 2, k = 1$; c is some constant d.

Since $a = b^k \implies 2 = 2^1$ (which is true), $T(n) \in \Theta(n^k logn)$.

With $k = 1$, the tight asymptotic bound on the program's runtime is $\Theta(nlogn)$.

Final Answer:
The tight asymptotic bound on the program's runtime is $\Theta(nlogn)$.

(b): Assume the lists are always very unevenly split; that is, $|L| = n - 2$ and $|G| = 1$ at each recursive call.

Tight Asymptotic Bound on the Runtime of Quicksort:
If $n - 2 \leq 1$, then $n \leq 3$.

So, for $n \leq 3$, $T(n) = c$, for some constant c, and the program runs in constant time with the tight bound $\Theta(1)$.

Otherwise, for $n > 3$, evaluate the closed-form for $T(n)$ as follows:

$$
\begin{aligned}
T(n) &= T(n-2) + [\overset{c}{\cancel{T(1)}} + dn] \\
&= T(\overset{(n-4)}{\cancel{(n-2)-2}}) + [\overset{c}{\cancel{T(1)}} + d(n-2) + [c + dn]] \\
&= T((n-4)-2) + \overset{c}{\cancel{T(1)}} + d(n-4) + [c + d(n-2) + [c + dn]] \\
&. \dots \\
&. \dots \\
&. \dots \\
&= c + d(kn - \sum_{i=0}^{k} 2i) + kc \\
&= d[kn - (2)(\frac{(k+1)(k)}{2})] + (k+1)c
\end{aligned}
$$

Through repeated substitution, $T(0)$ is eventually reached.

Notice that every unique call of $T(n)$ has the form $T(n - 2k)$, for some natural k.

Therefore, $T(0) = T(n-2k) \implies 0 = n-2k \implies k = \frac{n}{2}$; the call of $T(0)$ occurs when $k = \frac{n}{2}$.

Continue evaluating $T(n)$ with $k = \frac{n}{2}$:

$$\begin{aligned}
T(n) &= d[kn - (\cancel{2})(\frac{(\frac{n}{2}+1)(\frac{n}{2})}{\cancel{2}})] + (k+1)c \\
&= d[(\frac{n}{2})n - (\frac{n}{2}+1)(\frac{n}{2})] + (\frac{n}{2}+1)c \\
&= d[\frac{n^2}{2} - \frac{n^2}{4} - \frac{n}{2}] + (\frac{n}{2}+1)c \\
&= d[\frac{2n^2 - n^2 - 2n}{4}] + \frac{2cn+2c}{4} \\
&= \frac{dn^2 - 2dn + 2cn + 2c}{4}
\end{aligned}$$

Thus, $T(n) = \frac{dn^2-2dn+2cn+2c}{4}$ for $n > 3$.

Notice that in this closed-form, n^2 is the term with the highest degree, so it takes precedence.

Therefore, the tight asymptotic bound on the runtime of the program is $\Theta(n^2)$.

Final Answer:
The tight asymptotic bound on the runtime of the program is $\Theta(n^2)$.