Program Correctness

Hing Leung

Nov 19, 2010

1 Introduction

In this project, we'll learn how to prove the correctness of a program. We will read excerpts from the pioneering paper of Robert W. Floyd on "Assigning meanings to programs" (In Proceedings Symposium on Applied Mathematics, 19, Math. Aspects in Computer Science, pages 19–32, 1967.).

The topic of program correctness is often covered briefly in textbooks on data structures and algorithms. For example, program correctness are discussed in "Data Structures and Algorithms in Java" by M. T. Goodrich and R. Tamassia, John Wiley and Sons, p. 129-131, 2004, and in "Introduction to Algorithms" by T. Cormen, C. Leiserson, R. Rivest, and C. Stein, McGraw Hill, p. 17-19, 2003. However, the discussions tend to be very concise.

Another pioneering paper in program correctness is C. A. R. Hoare's "An axiomatic basis for computer programming" (Communications of the ACM, Vol 12, No. 10, pages 576–580, 1969). In advanced textbooks on the topic of program correctness, almost all of them are based on Hoare's axiomatic logic treatment in which mathematical axioms and inference rules are employed. As admitted by Hoare in his CACM paper, "The formal treatment of program execution presented it this paper is clearly derived from Floyd." (p. 583), and the treatment "is essentially due to Floyd but is applied to texts rather than flowchart" (p. 577). That is, conceptually Hoare's method is no different than Floyd's. However, Hoare's axiomatic approach, which is presented with the notations and terminology of axioms and inference rules, may be quite difficult for an upper level computer science undergraduate student. In practice, Hoare's technique, while based on an axioms and inference rules, is often carried out by annotating a program with assertions. In the appendix of this project, we give a brief overview of Hoare's method.

In Floyd's approach, programs are presented as flowcharts. One may wonder if flowchart programs are realistic, and whether Floyd's technique applies to programs that people write nowadays. In this project, we propose to replace flowchart programs by assembly-like programs with the use of "goto" statements. Each statement is accompanied by an assertion.

By adopting assembly-like programs as our preferred programming style, we are able to present the correctness proof technique in a simple and unified manner without the need

to revise the technique for each new programming language. We assume that a junior level computer science student is able to convert mechanically a program in a high level language into a assembly-like program. As the mechanical conversion preserves the same program execution behavior, we can establish the correctness of the original program in a high level language by proving the correctness of the derived assembly-like program.

In fact, a complete program correctness proof consists of two parts: a partial correctness proof and a termination proof. A partial correctness proof shows that a program is correct when indeed the program halts. However, a partial correctness proof does not establish that the program must halt. To prove a program always halt, the proof is called "termination proof". In this project, we focus on the partial correctness proof. That is, we are not going to cover the "termination proof".

Formal proof techniques are not only limited to proving program correctness. In 1994, when Intel first introduced the Pentium processor, a bug (known as Pentium FDIV bug) was found with the division operations in the floating point computation unit. Since then, automated theorem proving techniques have been employed extensively by Intel and AMD in integrated circuit design of computer processors.

2 Robert W. Floyd

Robert W Floyd [1] (June 8, 1936 – September 25, 2001) was an eminent computer scientist.

Born in New York, Floyd finished school at age 14. At the University of Chicago, he received a Bachelor's degree in liberal arts in 1953 (when still only 17) and a second Bachelor's degree in physics in 1958.

Becoming a computer operator in the early 1960s, he began publishing many noteworthy papers and was appointed an associate professor at Carnegie Mellon University by the time he was 27 and became a full professor at Stanford University six years later. He obtained this position without a Ph.D.

His contributions include the design of Floyd's algorithm, which efficiently finds all shortest paths in a graph, and work on parsing. In one isolated paper he introduced the important concept of error diffusion for rendering images, also called Floyd-Steinberg dithering (though he distinguished dithering from diffusion).

A significant achievement was pioneering the field of program verification using logical assertions with the 1967 paper Assigning Meanings to Programs. This was an important

[1] The biography is taken from wikipedia. (`http://en.wikipedia.org/wiki/Robert_W_Floyd`)

contribution to what later became Hoare logic.

Floyd worked closely with Donald Knuth, in particular as the major reviewer for Knuth's seminal book The Art of Computer Programming, and is the person most cited in that work. He was the co-author, with Richard Beigel, of the textbook The Language of Machines: an Introduction to Computability and Formal Languages, Freeman Publishing, 1994.

He received the Turing Award in 1978 "for having a clear influence on methodologies for the creation of efficient and reliable software, and for helping to found the following important subfields of computer science: the theory of parsing, the semantics of programming languages, automatic program verification, automatic program synthesis, and analysis of algorithms".

3 Floyd's method

First, we'll read the following excerpts from Floyd's paper:

> The basis of our approach is the notion of an interpretation of a program: that is, an association of a proposition with each connection in the flow of control through a program, where the proposition is asserted to hold whenever that connection is taken. To prevent an interpretation from being chosen arbitrarily, a condition is imposed on each command of the program. This condition guarantees that whenever a command is reached by way of a connection whose associated proposition is then true, it will be left (if at all) by a connection whose associated proposition will be true at that time. Then by induction on the number of commands executed, one sees that if a program is entered by a connection whose associated proposition is then true, it will be left (if at all) by a connection whose associated proposition will be true at that time. By this means, we may prove certain properties of programs, particularly properties of the form: "If the initial values of the program variables satisfy the relation R_1, the final values on completion will satisfy the relation R_2."
>
> DEFINITIONS. A *flowchart* will be loosely defined as a directed graph with a command at each vertex, connected by *edges* (arrows) representing the possible passages of control between the commands. An edge is said to be an *entrance* to (or an *exit* from) the command c at vertex v if its destination (or origin) is v. An *interpretation* I of a flowchart is a mapping of its edges on propositions. Some, but not necessarily all, of the free variables of these

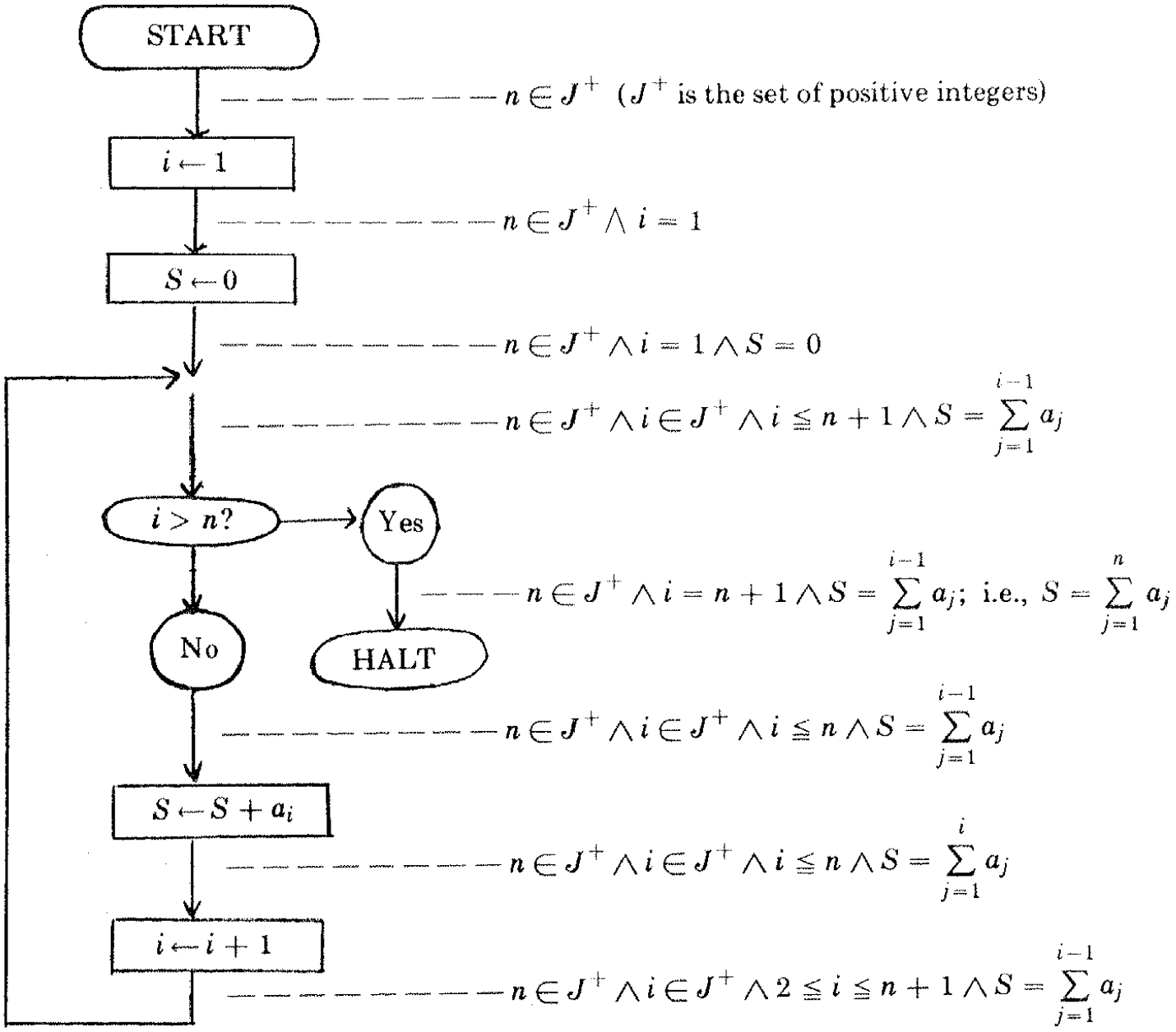

FIGURE 1. Flowchart of program to compute $S = \sum_{j=1}^{n} a_j$ ($n \geqq 0$)

> propositions may be variables manipulated by the program. Figure 1 gives an example of an interpretation. For any edge e, the associated proposition $I(e)$ will be called the *tag* of e. If e is an entrance (or an exit) of a command c, $I(e)$ is said to be an *antecedent* (or a *consequent*) of c.
>
> For any command c with k entrances and l exits, we will designate the entrances to c by $a_1, a_2, \ldots, a_k$, and the exits by $b_1, b_2, \ldots, b_l$. We will designate the tag of a_i by P_i $(1 \leq i \leq k)$, and that of b_i by Q_i $(1 \leq i \leq l)$. Boldface letters will designate vectors formed in the natural way from the entities designated by the corresponding nonboldface letters: for example, $\mathbf{P}$ represents $(P_1, P_2, \ldots, P_k)$.

In general, different executions of the same program on different inputs may give rise to very different execution paths of varying lengths. To reason that a program is correct (on all inputs), we need a proof method that covers all the different execution scenarios.

Floyd states that a verification (equivalently, partial correctness proof) of an interpretation of a flowchart needs not be too cumbersome. We only need to check

> for each command c of the flowchart, if control should enter the command by an entrance a_i with P_i true, then control must leave the command, if at all, by an exits b_j with Q_j true.

Task 1: Suppose it has been established that every command c in the flowchart is "correct/verified" with respect to the antecedents and consequents. By using a mathematical induction on the number of steps of the execution of a flowchart program, justify that interpretation of the flowchart program is "correct", in the sense that the tagged propositions are always observed.

But, how can we do verification for each command c in a flowchart program? Are we doing verification in an ad-hoc manner? Or, can there be a mechanical/systematic way for doing verification for each command type?

Next, Floyd introduces the concept of a *verification condition* on the antecedents and consequents of a command c. Given that a command c has antecedents $\mathbf{P} = (P_1, P_2, \ldots, P_k)$ and consequents $\mathbf{Q} = (Q_1, Q_2, \ldots, P_l)$, the verification condition, denoted by $V_c(\mathbf{P}; \mathbf{Q})$, gives the verification needed to establish that

> if control should enter the command c by an entrance a_i with P_i true, then control must leave the command, if at all, by an exits b_j with Q_j true.

It is natural to require that the verification conditions developed satisfy the following:

> The verification condition must be so constructed that a proof that the verification condition is satisfied for the antecedents and consequents of each command in a flowchart is a verification of the interpreted flowchart.

Floyd developes the verification condition for the assignment statement in a series of steps. The verification condition[2] obtained involves the use of existential quantifier. We give another formulation of the verification condition for the assignment operator as follows:

The assignment statement $x \leftarrow f(x, \mathbf{y})$ is verified for antecedent $P_1(x, \mathbf{y})$ and consequent $Q_1(x, \mathbf{y})$ if we can show that

$$P_1(x, \mathbf{y}) \rightarrow Q_1(f(x, \mathbf{y}), \mathbf{y})$$

Task 2: Give an informal justification for the verification condition for assignment operator.

To compute $Q_1(f(x, \mathbf{y}), \mathbf{y})$, we substitute every occurrence of x in $Q_1(x, \mathbf{y})$ by the expression $f(x, \mathbf{y})$ given in right hand side of the assignment statement $x \leftarrow f(x, \mathbf{y})$. The technique is called back substitution.

The verification conditions for other statement types are quite straight forward.

Consider the branch statement type. Suppose c is the branch (conditional) statement with antecedent P_1, conditional test Φ, consequent Q_1 when Φ is true, and consequent Q_2 when Φ is false. The verification condition $V_c(P_1; Q_1, Q_2)$ for the branch statement is

$$((P_1 \wedge \Phi) \rightarrow Q_1) \wedge ((P_1 \wedge \neg\Phi) \rightarrow Q_2)$$

Task 3: Give an informal justification for the verification condition for branching operator.

In a flowchart program, there is a possibility that several branches are joining and merging into one branch. In Figure 1, two branches are joining before the statement that performs the test of whether $i > n$; one branch flows down from the statement $S \leftarrow 0$, with another branch coming from the lower part of the program after the assignment statement $i \leftarrow i + 1$.

We consider the joining statement to be a dummy statement with antecedents $\mathbf{P}$ and consequent Q_1. If there are two branches joining, then $\mathbf{P} = (P_1, P_2)$. Floyd gives the

[2] In Floyd's paper, in the context of deductive system with axioms and inference rules, the verification condition is actually given as follows: If P_1 has the form $R(x, \mathbf{y})$ and if $(\exists x_0)(x = f(x_0, \mathbf{y}) \wedge R(x_0, \mathbf{y})) \vdash Q_1$, then $V_{x \leftarrow f(x,\mathbf{y})}(P_1, Q_1)$.

verification condition[3] $V_c(P_1, P_2; Q_1)$ for the joining statement as

$$(P_1 \rightarrow Q_1) \wedge (P_2 \rightarrow Q_1)$$

Task 4: Give an informal justification for the verification condition for joining operator.

In Figure 1, we see that there are two other special statements. One is the "start" statement and the other one is the "halt" statement. The tag (proposition) for the edge after the start statement is given as

$n \in J^+$ (J^+ is the set of positive integers)

The condition that "n is a positive integer" is assumed to hold for the input. Thus, the verification condition for the start statement is considered to be true. On the other hand, the tag (proposition) for the edge before the halt statement is

$n \in J^+ \wedge i = n + 1 \wedge S = \sum_{j=1}^{i-1} a_j$; i.e., $S = \sum_{j=1}^{n} a_j$

The tag before the the halt statement is what the program is supposed to achieve, a goal that we are not going to dispute. As in the case with the start statement, the verification condition for the halt statement is again considered to be true.

4 Program Summation

Figure 1 gives an example flowchart program taken from Floyd's paper for computing $S = \sum_{j=1}^{n} a_j$ where $n \geq 0$. Corresponding to the flowchart program, we construct an equivalent while-loop program and an equivalent assembly program[4] as follows:

```
i = 1
S = 0
while (i <= n) {
   S = S + a_i
   i = i + 1
}
```

[3]Actually, Floyd gives the verification condition as $(P_1 \vee P_2) \rightarrow Q_1$, which is equivalent to $(P_1 \rightarrow Q_1) \wedge (P_2 \rightarrow Q_1)$ according to mathematical logic.

[4]For each flowchart program, it is easy to construct an equivalent assembly program. We favor assembly programs over flowchart programs as we find it easier to give references to the differenct edges in an assembly program than in a flowchart program.

```
1.   <P1>    i = 1
2.   <P2>    S = 0
3.   <P3>
4.   <P4>    if (i > n) goto 8
5.   <P5>    S = S + a_i
6.   <P6>    i = i + 1
7.   <P7>    goto 4
8.   <P8>    Halt
```

P1 $\equiv$ $n \in J^+$ (J^+ is the set of positive integers)
P2 $\equiv$ $n \in J^+ \wedge i = 1$
P3 $\equiv$ $n \in J^+ \wedge i = 1 \wedge S = 0$
P4 $\equiv$ $n \in J^+ \wedge i \in J^+ \wedge i \leq n + 1 \wedge S = \sum_{j=1}^{i-1} a_j$
P5 $\equiv$ $n \in J^+ \wedge i \in J^+ \wedge i \leq n \wedge S = \sum_{j=1}^{i-1} a_j$
P6 $\equiv$ $n \in J^+ \wedge i \in J^+ \wedge i \leq n \wedge S = \sum_{j=1}^{i} a_j$
P7 $\equiv$ $n \in J^+ \wedge i \in J^+ \wedge 2 \leq i \leq n + 1 \wedge S = \sum_{j=1}^{i-1} a_j$
P8 $\equiv$ $n \in J^+ \wedge i = n + 1 \wedge S = \sum_{j=1}^{i-1} a_j$; i.e., $S = \sum_{j=1}^{n} a_j$

The proposition Pi (where $1 \leq i \leq 8$) is the antecedent of statement/command i. Note that the propositions considered in the assembly program are the same as the propositions considered in the flowchart program.

To verify the given interpretation, we need to check to see if the verification condition is satisfied for every statement in the flowchart/assembly program.

Task 5. What is the verification condition for statement 1? Show that the verification condition for statement 1 is satisfied. Hint: the verification condition involves P1, statement 1 and P2.

Task 6. What is the verification condition for statement 4? Show that the verification condition for statement 4 is satisfied. Hint: the verification condition involves P4, statement 4, P5 and P8.

Task 7. What is the verification condition for statement 6? Show that the verification condition for statement 6 is satisfied. Hint: the verification condition involves P6, statement 6, and P7.

Task 8. Verify that the verification conditions for all the rest of the statements are satisfied.

5 Program Square Root

Consider the following program[5] which computes the square root b of a given nonnegative integer a. Note that b is a nonnegative integer such that $b^2 \leq a < (b+1)^2$.

```
b = 0
c = 1
d = 1
while (c <= a) {
   b = b+1
   d = d+2
   c = c+d
}
```

The assembly codes equivalent are as follows:

```
1.   <P1>    b = 0
2.   <P2>    c = 1
3.   <P3>    d = 1
4.   <P4>    if (c > a) goto 9
5.   <P5>    b = b+1
6.   <P6>    d = d+2
7.   <P7>    c = c+d
8.   <P8>    goto 4
9.   <P9>
```

It is given that
P1 $\equiv$ $a \geq 0$
P9 $\equiv$ $(b \geq 0) \wedge (b^2 \leq a) \wedge (a < (b+1)^2)$
By proving that P9 holds when the program halts, we show that b is the integral part of the square root of a.

To help you to tackle the correctness proof, the loop invariant P4 is given below:
P4 $\equiv$ $(b \geq 0) \wedge (b^2 \leq a) \wedge (c = (b+1)^2) \wedge (d = 2b+1)$.

Task 9. What are P2, P3, P5, P6, P7, P8? Verify that the verification conditions for all the statements are satisfied. (Hint: First, decide what to choose for P8. Then use back substitutions to determine other assertions.)

[5]The program is taken from page 4 of the monograph "Lecture on the logic of computer programming" by Zohar Manna, CBMS-NSF regional conference series in applied mathematics, 1980.

6 Program Chocolate Bars

Suppose we can buy a chocolate bar from the retail store for $1 each. Inside every chocolate bar, there is a coupon included. With 10 coupons, one can redeem for one chocolate bar. Again, a coupon is found in the chocolate bar redeemed. The following program computes the number of chocolate bars that one can consume with n dollars:

```
// b denotes the number of chocolate bars
// c denotes the number of coupons
b = n
c = n
while (c >= 10) {
   c = c-9
   b = b+1
}
```

Interestingly, the final answer for b can be computed by the arithmetic expression $n + \lfloor \frac{n-1}{9} \rfloor$ where $\lfloor x \rfloor$, called the floor of x, denotes the largest integer not greater than x. In most programming languages, $n + \lfloor \frac{n-1}{9} \rfloor$ can be written simply as $n + (n-1)/9$. The above loop program can be re-written as the following assembly program:

```
1.   <P1>    b = n
2.   <P2>    c = n
3.   <P3>    if (c < 10) goto 7
4.   <P4>    c = c-9
5.   <P5>    b = b+1
6.   <P6>    goto 3
7.   <P7>
```

It is given that

P1 $\equiv$ $n > 0$

P7 $\equiv$ $b = n + \lfloor \frac{n-1}{9} \rfloor$

The loop invariant P3 is given as follows:

P3 $\equiv$ $\left(b + \lfloor \frac{c-1}{9} \rfloor = n + \lfloor \frac{n-1}{9} \rfloor\right) \ \wedge \ (c > 0)$

Task 10. What are P2, P4, P5, P6 ? Verify that the verification conditions for all the statements are satisfied. (Hint: First, decide what to choose for P6. Then use back substitutions to determine other assertions.)

7 Program Partition

In page 146 of textbook "Introduction to Algorithms", 2nd edition by Cormen, Leiserson, Rivest and Stein, the codes for the `Partition` algorithm for Quicksort is given as follows:

```
Partition(A,p,r)

  x = A[r]
  i = p-1
  for j = p to r-1
      if (A[j] <= x) {
         i = i+1
         exchange A[i] and A[j]
      }
  exchange A[i+1] and A[r]
  return i+1
```

We can re-write the program into an equivalent assembly program as follows:

```
1.   <P1>    x = A[r]
2.   <P2>    i = p-1
3.   <P3>    j = p
4.   <P4>    if (j  > r-1) goto 10
5.   <P5>    if (A[j] > x) goto 8
6.   <P6>    i = i+1
7.   <P7>    exchange A[i] and A[j]
8.   <P8>    j = j+1
9.   <P9>    goto 4
10.  <P10>   exchange A[i+1] and A[r]
11.  <P11>   q = i+1
12.  <P12>
```

It is given that
`P1` $\equiv p \leq r$
and
`P12` $\equiv (\forall k, (p \leq k \leq q-1) \rightarrow (A[k] \leq x)) \wedge (\forall k, (q+1 \leq k \leq r) \rightarrow (A[k] > x)) \wedge (A[q] = x) \wedge (p \leq q \leq r)$

Technically, it is *not* accurate to say that the partition algorithm is correct just by showing that `P12` holds when the program halts. A more careful proof needs to argue

that the resulting array (from index p to index r) is a re-arrangement (or, permutation) of the original array from the same set of indices. However, one can easily see that this is true since the program limits itself in performing swapping of array entries from index p to index r. The fact that we are only swapping array entries between index p and index r is going to be part of the proof that you are constructing. That is, in `P7`, we have to show that $p \leq i \leq j \leq r$. Similarly, in `P10`, we also have to show that $p \leq i+1 \leq j \leq r$.

In page 146 of the textbook, a proposition is given for the loop invariant, which translate to the following proposition for `P4`:
$(\forall k,\ (p \leq k \leq i) \rightarrow (A[k] \leq x)) \wedge (\forall k,\ (i+1 \leq k \leq j-1) \rightarrow (A[k] > x)) \wedge (A[r] = x) \wedge (p-1 \leq i < j \leq r)$

Task 11. What are `P2, P3, P5, P6, P7, P8, P9, P10, P11` ? Verify that the verification conditions for all the statements are satisfied.

8 Remarks

In proving the correctness of a program, the most challenging part is to give the assertion (called the loop invariant) that associates with the while-statement (or, for-statement). In the four programs considered in this project, with the loop invariants given, the rest of the assertions can be determined in a rather systematic and mechanical manner.

Appendix (Hoare's Method)

Floyd's method considers the verification condition of statement for flowchart/assembly program. Hoare's method explains how to verify program properties for high level programming languages which statement types include while-statement and if-statement.

Notation: Let `S` be a block of statements (or, a program). We write

$$\{P\}\ \texttt{S}\ \{Q\}$$

if given the antecedent P, the consequent Q holds after the execution of `S`.

Assignment Rule

From the verification condition for assignment statement $x \leftarrow f(x, \mathbf{y})$ of Floyd's method, it is clear that the following holds

$$\{Q(f(x, \mathbf{y}), \mathbf{y})\}\ x \leftarrow f(x, \mathbf{y})\ \{Q(x, \mathbf{y})\}$$

If Rule

Consider the if-statement

$$\texttt{if (c) then S1 else S2}$$

Suppose we want to establish

$$\{P\}\ \texttt{if (c) then S1 else S2}\ \{Q\}$$

where P is the antecedent and Q is the consequent. Given that P is the antecedent condition for the if-statement, both P and `c` hold when the program starts executing `S1`. Similarly, P and $\neg$`c` hold when the program begins executing `S2`. In order the consequent Q holds for the if-statement, it is required that both

$$\{P \wedge \texttt{c}\}\ \texttt{S1}\ \{Q\}$$

and

$$\{P \wedge \neg\texttt{c}\}\ \texttt{S2}\ \{Q\}$$

are true. To summarize, given that

$$\{P \wedge \texttt{c}\}\ \texttt{S1}\ \{Q\},\ \{P \wedge \neg\texttt{c}\}\ \texttt{S2}\ \{Q\}$$

it follows that

$$\{P\}\ \texttt{if (c) then S1 else S2}\ \{Q\}$$

As a special case when `S2` is a dummy statement, given that

$$\{P \wedge \texttt{c}\}\ \texttt{S1}\ \{Q\}, (P \wedge \neg\texttt{c}) \rightarrow Q$$

the following holds:

$$\{P\}\ \texttt{if (c) then S1}\ \{Q\}$$

While Rule

Consider the while-statement

$$\texttt{while (c) S}$$

where `c` is the condition and `S` is the body of the loop. Suppose I is the proposed loop invariant. The antecedent condition for `S` is $I \wedge \texttt{c}$ as both I and `c` holds when the program starts executing `S`. In order that the loop invariant holds again after the execution of `S`, it is necessary that I is the consequent of `S`. Thus,

$$\{I \wedge \texttt{c}\}\ \texttt{S}\ \{I\}$$

By picking the antecedent for the while statement as I, and given that $\{I \wedge \texttt{c}\}\ \texttt{S}\ \{I\}$ holds, the loop invariant I holds and the consequent for the while-statement is thus $I \wedge \neg\texttt{c}$. To summarize, given that

$$\{I \wedge \texttt{c}\}\ \texttt{S}\ \{I\}$$

is verified, the following holds about the while-statement:

$$\{I\}\ \texttt{while (c) S}\ \{I \wedge \neg\texttt{c}\}$$

Consequence Rule

It should be clear that given

$$\{P\}\ \texttt{S}\ \{Q\},\ P' \rightarrow P,\ Q \rightarrow Q'$$

then the following holds

$$\{P'\}\ \texttt{S}\ \{Q'\}$$

Composition Rule

It should be clear that given

$$\{P\}\ \texttt{S1}\ \{Q\},\ \{Q\}\ \texttt{S2}\ \{R\}$$

then the following holds

$$\{P\}\ \texttt{S1;S2}\ \{R\}$$

Task 12. Using Hoare's method for while-loop programs, re-do the partial correctness proofs for Program Square Root, Program Chocolate Bars and Program Partition. Hint: Program Partition needs to be re-written using while-loop instead of for-loop.

9 Notes to the Instructor

The project is designed to focus on the key ideas in program correctness, rather than on the formal aspects of the theory. Terminologies like axioms, inferences, and deductive systems are deliberately avoided. Emphasis is placed on the practice of proving the partial correctness of interesting programs. It is the author's belief that the students will appreciate better the power of program correctness through the analysis of seemingly simple and yet non-trivial programs.

The correctness proof of Program Partition (Section 7) is very challenging. It may be skipped if time is not permitted, or if the students are finding it too difficult.