Chapter 1

Correctness of Programs

Written by David Gries

Table of Contents

1.1 Assertions

Let us call any true-false statement a *predicate*. For example, $x > y$ is a predicate —it is either true or false in any machine state that gives arithmetic values to x and y. Another example is "n is the number of negative values in array $b[1..10]$". A predicate can be written in English or mathematics or Japanese or any combination thereof.

An *assertion* is a predicate enclosed within braces { and } within a program to indicate that the predicate is true at that point; it is asserted that the predicate is true there. Note that the braces are *not* part of the assertion; they are placed around the asssertion only to delimit it.

The following notation, called a "triple", is also a predicate:

$\{Q\}\ S\ \{R\}$

where Q and R are predicates and S is a statement. Q is called the precondition of S and R the *post-condition*. This triple is equivalent to the following English sentence:

> Execution of statement S begun in a state in which Q is true is guaranteed to terminate, and R is true in the final state.

Here are three examples, all of which are true.

$\{x < 0\}\ x = \ x + 1;\ \{x < 1\}$

This says that execution of $x = x + 1;$ with $x < 0$ terminates with $x < 1$.

$\{true\}$ **if** $(x \leq y)\ \ z = \ x;$
else $z = \ y;$
$\{z = min(x, y)\}$

This says that execution of the conditional statement beginning in any state (*true* represents the set of all states) terminates with x equal to the minimum of x and y.

$\{false\}$ **while** $(true)$ **do** $\{true\}$

This example is more subtle. Since the loop never terminates, it certainly cannot terminate in any state. The predicate *false* stands for the set of no states; *false* is always false.

The above predicate is always confusing at first. For *any* statement S and predicate R, the triple

$\{false\}\ \ S\ \ \{R\}$

is false only when some state s (say) satisfies the precondition and execution of S in s either does not terminate or terminates with R false. Since no such state s satisfies the precondition, the triple is true.

Note that a triple $\{Q\}\ \ S\ \ \{R\}$ says *absolutely nothing* about S executed in a state in which Q is false. It says only something about execution begun when Q is true.

Any program can be specified by giving a pre- and post-condition pair. Here are three examples:

Specification of an algorithm to sort $b[m..n]$:

precondition : $m \leq n-1$

postcondition : $b[m..n]$ is a permutation of its initial value **and** $b[m] \leq b[m+1] \leq \cdots \leq b[n]$

Specification of an algorithm to read 10 integers and print their sum:

precondition : the input contains at least 10 integers $a_0, \ldots, a_9$

postcondition : the input contains 10 fewer integers **and** output contains $a_0 + \cdots + a_9$

Specification of an algorithm to find an integer approximation to the square root of an integer:

pre : $0 \leq b$

post : $a^2 \leq b \leq (a+1)^2$

We *annotate* a program by inserting assertions to help the reader understand what is true at various points of the program. A program with assertions in it is called an *annotated* program.

When two assertions appear adjacent within a program, the second is a consequence of the first. We often insert the word "hence" at the beginning of the second assertion to emphasize this point.

Examples In these examples, assume that all variables have type integer.

$x = 2; \ \{x = 2\}$

This indicates that x has the value 2 after execution of $x = 2$.

$x = 2; \ \{x = 2\} \ y = 10; \ \{x * y = 20\}$

This indicates that x has the value 2 after execution of the first statement and that $x * y$ equals 20 after execution of the second.

$\{y > 0$ **and** $x \geq 0\} \ z = x$ **div** $y; \ \{0 \leq z \leq x\}$

This says that *if* $y > 0$ **and** $x \geq 0$, then execution of $z = x$ **div** y terminates with $0 \leq z \leq x$. It says nothing about execution of the statement when $y = 0$ (a runtime error, division by 0 , would occur) or if $y < 0$.

$\{x = X$ **and** $y = Y\}$

Swap x *and* y;

$\{x = Y$ **and** $y = X\}$

This example illustrates the use of names for initial or final values of variables. These names do not belong in the program, but only in its description. Here is the statement rendered in English: "Under the condition that x contains a value X and y a value Y , after execution of *Swap* x *and* y;, x contains Y and y contains X ."

Here's the last example:

$\{x \leq 0\}$

{hence, $x \leq 5$}

$x = x + 20;$

$\{x \leq 25\}$

{hence, $x \leq 100$}

1.2 The Assignment Statement

Consider an assignment statement $x = e;$, where x is a simple variable. Suppose its execution is supposed to truthify a postcondition R (i.e. make R come true). How do we determine in what initial states that will happen? There is a simple way to determine the precondition. First, a definition:

R_e^x stands for a copy of R in which all occurrences of x are replaced by e.

Examples

(0) $(x = y)_w^x = (w = y)$

(1) $(x = y)_{x+2}^x = (x + 2 = y)$

(2) $(x = x + 2)_{z+x+y}^x = (z + x + y = z + x + y + 2)$

We assert that the following always holds:

$\{R_e^x\} \ x = e; \ \{R\}$

Actually, R is true after execution of $x = e;$ if and only if R_e^x is true before. (Of course, we assume that evaluating e will not result in a runtime error, such as division by 0 or subscript out of range.)

Examples

(0) $\{2 = 2\} \ x = 2; \ \{x = 2\}$

Since $2 = 2$ is true, this is equivalent to $\{true\} \ x = 2; \ \{x = 2\}$.

(1) $\{x + 1 \geq 0\} \ x = x + 1; \ \{x \geq 0\}$

The precondition can be written as $x \geq -1$.

(2) $\{(x - y) * y + y * y = 5\}$
$x = x - y;$
$\{x * y + y * y = 5\}$

The precondition can be written as $x * y = 5$.

1.3 The Conditional Statement

Consider the following conditional statement, which sets z to the maximum of x and y:

if $(x \geq y)$ $z = x$;
else $z = y$;

We annotate it fully as follows:

$\{true\}$
if $(x \geq y)$ $\{true$ and $x \geq y\}$
$\{$hence, $x = max(x, y)\}$
$z = x$;
$\{z = max(x, y)\}$
else $\{true$ and $x < y\}$
$\{$hence, $y = max(x, y)\}$
$z = y$;
$\{z = max(x, y)\}$
$\{z = max(x, y)\}$

Rarely will we annotate a program in such detail; it is done here only to show you a fully annotated program. Let us discuss a few points.

- The precondition of the then-part of an IF is a combination of the precondition of the IF and the if-condition; it simply states what is known at that point based upon what is known before the IF.
- The precondition of the else-part of an IF is a combination of the precondition of the IF and the falsity of the if-condition; it simply states what is known at that point based upon what is known before the IF.
- Note the use of two assertions together; the first implies the second.
- The postcondition of an IF is the "or" of the postconditions of the then-part and else-part. In this case, the two parts have the same postcondition.

1.4 The loop

Consider the following loop, which stores in variable s the sum of the elements of array $b[1..10]$:

$i = 1$; $s = b[1]$;
while $(i \neq 10)$ {
$i = i + 1$;
$s = s + b[i]$;
}

How can we understand it? Is there a general way to look at any loop that provides understanding? The answer is yes. We illustrate the method on this loop and then give the general method. Let us annotate the program, where we use the predicate P to describe the values of s and i before (and after) each iteration:

P: $1 \leq i \leq 10$ **and**
s is the sum of first i elements of b.

Here is the annotated loop.

$\{true\}$
$i = 1$; $s = b[1]$;
$\{P\}$
while $(i \neq 10)$ {
$\{P$ and $i \neq 10\}$
$i = i + 1$;
$s = s + b[i]$;
$\{P\}$
}
$\{P$ and $i = 10\}$
$\{$hence, s is sum of $b[1..10]\}$

Make sure that you understand each assertion, i.e. check that it is true at the point of execution where it is placed. One rule to use here is that if P is true after one iteration of the loop (after execution of the loop body) then it is true before the next iteration. This should be obvious.

Predicate P is called an *invariant relation* of the loop, because it is invariantly true before and after each iteration. Predicate P is nothing more than a *definition of variables i and s* that holds before and after each iteration.

We must also be sure that the loop terminates. We do this by giving an expression t (just to have a name for it) that gives an upper bound on the number of iterations still to be performed and show that it decreases with each iteration. In this case, for t we can use the expression $10 - i$, which gives exactly the number of iterations still to be performed.

Expression t is called a *bound function* of the loop.

We now state what must be done to prove that

$\{Q\}$ **while** (B) S $\{R\}$

holds, using an *invariant* P and *bound function* t.

- Prove that P is true before the loop: prove that Q implies P.
- Prove that P is indeed a loop invariant: prove $\{P$ and $B\}$ S $\{P\}$.

- Prove that upon termination R holds: prove that P and $\neg B$ implies R.
- (3) Prove that t decreases with each iteration.
- Prove that t is indeed an upper bound on the number of iterations. This means: if there is another iteration to perform, then $t > 0$, i.e. P and B implies $t > 0$.

A loop is usually annotated as shown below. This format makes clear what the invariant and bound function are and allows the invariant to appear only once.

$\{inv: \ P\}$
$\{bd: \ t\}$
while (B) S

Here are two annotated loops. Practice understanding them using the five rules given above. Get in the habit of viewing a loop only in terms of whether these five rules hold or not.

(0) This first program finds the first position i of value x in array $b[1..100]$, given that x is in b. The specification is given by Q and R, the pre- and postconditions.

$\{Q: \ x$ is somewhere in $b[1..100]\}$
$i = 1;$
$\{inv: \ x \notin b[1..i-1];$ hence, $x \in b[i..100]\}$
$\{bd: \ 100 - i\}$
while $(b[i] \neq x)$ $i = i + 1;$
$\{R: \ b[i] = x\}$

(1) This second example calculates the quotient q and remainder r when x is divided by y. For $y > 0$ and $x \geq 0$, q and r are defined (uniquely) by the predicate

$R: \quad x = q * y + r \quad$ and $\quad 0 \leq r < y$

$\{x \geq 0 \ \text{ and } \ y > 0\}$
$q = 0; \ r = x;$
$\{Q: \ x \geq 0 \ \text{ and } \ y > 0 \ \text{ and } \ q = 0 \ \text{ and } \ r = x\}$
$\{inv: \ x = q * y + r \ \text{ and } \ 0 \leq r\}$
$\{bd: \ r \ \textbf{div} \ y\}$
while $(r \geq y)\{ \ r = r - y; \ q = q + 1; \ \}$
$\{R\}$

1.5 Developing a loop from an invariant

We state a general method for developing a loop when given a precondition Q, a postcondition R, an invariant P, and a bound function t. There are three steps to this:

- Find initialization that makes P true.
- Find the loop condition as follows: find a predicate NB such that from P and NB we can conclude that R is true. Take as loop condition the predicate $\neg NB$.
- Develop the body of the loop as follows:

  (a) Get an initial version of the loop body by finding a way to make progress (i.e. get closer to termination, decrease bound function t).

  (b) Modify the loop body so that P is true after its execution.

Example

Consider writing a program to store zeros in even positions of array $b[1..n]$. We are given the following:

Precondition $Q1: \quad 0 \leq n$
Postcondition $R: \quad b[2], b[4], \ldots, b[2 * (n \ \textbf{div} \ 2)]$ are zero
Invariant $P: \quad 2 \leq i \leq n + 2$ and i is even and $b[2], b[4], \ldots, b[i-2]$ are zero
Bound function $t: \ n + 1 - i$

Step 1. We can truthify P with $i = 2;$.
Step 2. If P is true, and if $i > n$, then R is true. Hence, choose NB to be $i > n$. So condition B is $i \leq n$.
Step 3. To make progress, add 2 to i. But before this is done, 0 has to be stored in $b[i]$.

Hence, the program is

$i = 2;$
$\{inv: \ P$ (given above)$\}$
$\{bd: \ t$ (given above)$\}$
while $(i \leq n)$ $\{ \ b[i] = 0; \ i = i + 2; \ \}$

1.6 Exercises (with solutions)

Write algorithms for the following problems. The solutions are in Sec. 1.7.

(0) **Linear search**.

Given is an array $b[m..n]$, where $m \leq n+1$. Given is a value x, which may or may not be in $b[m..n]$. We want an algorithm that stores in i the index of x in $b[m..n]$, if $x \in b[m..n]$, and stores $n+1$ in i otherwise. Here is the precondition Q, postcondition R, invariant P, and bound function t for a loop:

Q: $m \leq n+1$
R: $m \leq i \leq n+1$ and
$x \notin b[m..i-1]$ and
$(i = n+1 \vee x = b[i])$
P: $m \leq i \leq n+1$ and
$x \notin b[m..i-1]$
t: $n+1-i$

(1) **Linear search** (same problem as (0) but a different invariant and thus a different algorithm).

Given is an array $b[m..n]$, where $m \leq n+1$. Given is a value x, which may or may not be in $b[m..n]$. We want an algorithm that stores in i the index of x in $b[m..n]$ if $x \in b[m..n]$ and stores $n+1$ in i otherwise. Here is the precondition Q and postcondition R, as well as the invariant P and bound function t, which uses a fresh integer variable f:

Q: $m \leq n+1$
R: $m \leq i \leq n+1$ and
$x \notin b[m..i-1]$ and
$(i = n+1 \vee x = b[i])$
P: $m \leq i \leq f \leq n+1$ and
$x \notin b[m..i-1]$ and
$(f = n+1 \vee x = b[i])$
t: $f - i$

(2) **Linear search with sentinel**.

Write a loop (with initialization) for the following problem. Given is a value x, which may or may not be in $b[m..n]$. We want an algorithm that stores in i the index of x in $b[m..n]$ if $x \in b[m..n]$ and stores $n+1$ in i otherwise. Array element $b[n+1]$ may be changed.

The difference between this exercise and exercise (0) is that here we can use position $b[n+1]$ of the array to contain the value we are looking for —it can act as a sentinel to stop the search. Note how the invariant requires $x = b[n+1]$, and, since this is not true initially, the initialization must store x in $b[n+1]$.

Here is the precondition Q, postcondition R, and invariant P and bound function t for a loop:

Q: $m \leq n+1$
R: $m \leq i \leq n+1$ and
$x \notin b[m..i-1]$ and
$(i = n+1 \vee x = b[i])$
P: $m \leq i \leq n+1$ and
$x \notin b[m..i-1]$ and
$x = b[n+1]$
t: $n+1-i$

(3) **Binary search**.

Given is a sorted array $b[1..n]$ —i.e. $b[1] \leq b[2] \leq b[3] \leq \cdots \leq b[n]$. We want to search it for a value x.

The assertions concerning the algorithm are written assuming the existence of two "virtual" elements $b[0] = -\infty$ and $b[n+1] = +\infty$. These two array elements don't exist, but assuming they do allows us to write the specification and assertions more easily. The algorithm won't refer to them, of course.

If $x \in b$, the algorithm finds the rightmost occurrence of x; if $x \notin b$, it finds the position after which x should be inserted (but doesn't insert it).

For example, for $b[1..6] = (2,4,4,4,4,5)$ we give below various values of x and the corresponding resulting position i:

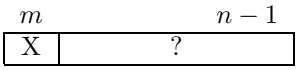
x:	1	2	3	4	5	6
i:	0	1	1	5	6	6

Here is the given information:

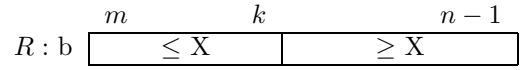
Q: $0 \leq n$
R: $0 \leq i \leq n$ and
$b[i] \leq x < b[i+1]$
P: $0 \leq i < j \leq n+1$ and
$b[i] \leq x < b[j]$
t: $j - i - 1$

(4) **Partition**.

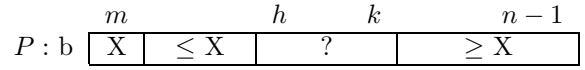
Given is array segment $b[m..n-1]$, with $n > m$, that satisfies



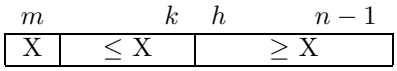
(Here, X is NOT a variable of the program; it is just a name that is used to denote the value initially in $b[m]$. X may *not* appear in the program.) We want an algorithm to permute (rearrange) the values of b and store a value in k to truthify



The loop invariant P is to be



The loop has as its postcondition



Since R' is different from R, some final statements have to follow the loop. The bound function t is $k - h + 1$.

Write the algorithm as a procedure with the Java heading

```
// Given b[m..n-1] nonempty, truthify R
// (see above) and return k
public static int partition(int [] b,
                            int m, int n)
```

(5) **Saddleback search**.

A value x is known to be in array $b[1..m, 1..n]$. Further, each row of b and each column of b is in ascending order. Store in i and j a position of x in b. The precondition Q and postcondition R are

Q : $x \in b[1..m, 1..n]$ and
each row of b is ordered and
each col of b is ordered

R : $x = b[i, j]$, which can be written as
$x \in b[i..i, j..j]$.

The invariant and bound function of the loop of the algorithm are

inv : $1 \le i \le m$ and
$1 \le j \le n$ and
$x \in b[i..m, 1..j]$

bd : $j + m - i$

Write the algorithm.

(6) **Insertion sort**.

Write an algorithm to sort array $b[m..n]$. The particular invariant we choose for the main loop leads to an algorithm that is known as *Insertion sort*. Here is the information:

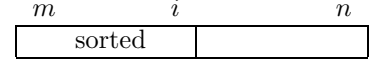
Q : $m \le n$

R : b is a permutation of its initial value and
$b[m] \le b[m+1] \le \cdots \le b[n]$

P : $m \le i \le n$ and $b[m..i]$ is sorted

t : $n - i$

We can write invariant P as:



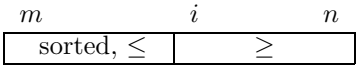
(7) **Selection sort**.

Write an algorithm to sort array $b[m..n]$. The invariant we choose leads to an algorithm called *Selection sort*. Here is the information:

Q : $m \le n$

R : b is a permutation of its initial value and
$b[m] \le b[m+1] \le \cdots \le b[n]$

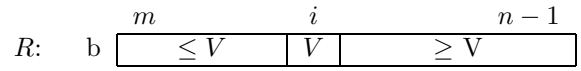
P : $m \le i \le n$ and



t : $n - i + 1$

(8) **Find**.

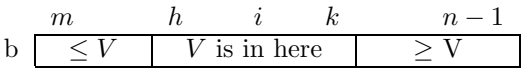
Given is an integer array segment $b[m..n-1]$ and an integer i satisfying $m \le i < n$. Find a value V (say) and permute $b[m..n-1]$ so that the following is truthified:



Note that if $i = m$ then the smallest value will be stored in the first position $b[m]$; if $i = n - 1$ the largest value will be stored in the last position $b[n-1]$, and if $i = (m + n)$ **div** 2 then the median is stored in $b[i]$. Thus, the algorithm can be used to find the value that belongs in any position of the array.

Here is the loop invariant for the loop of the algorithm:

P : $m \le h \le i \le k < n$ and



The bound function is $k + 1 - h$.

Here are some hints. First, the value V is not known until the loop terminates; we just use V as a name for the value to be found. Second, each iteration of the loop should call function *partition* of example (4) above. Third, it is not necessary to sort the array completely; it is only necessary to permute it until it can be recognized that V is in $b[i]$.

1.7 Sample solutions

(0) (**Linear search**)

```
i = m;
while (i ≠ n + 1 && x ≠ b[i]) i = i + 1;
```

(1) (**Linear search**)

```
i = n; f = n + 1;
while (i ≠ f) {
  if (x ≠ b[i]) i = i + 1;
  else f = i;
}
```

(2) (**Linear search with sentinel**)

```
i = m; b[n + 1] = x;
while (x ≠ b[i]) i = i + 1;
```

(3) (**Binary search**)

```
i = 0; j = n + 1;
{inv : 0 ≤ i < j ≤ n + 1 and
       b[i] ≤ x < b[j]}
{bound function : j − i − 1}
while (j ≠ i + 1) {
  int e = (i + j) div 2;
  {i < e < j}
  if (b[e] ≤ x) i = e;
  else j = e;
}
```

(4) (**Partition algorithm**)

```
public static int partition(int [] b,
                            int m, int n);
int h = m + 1; int k = n − 1; int x = b[m];
{inv : P (given in problem)}
while (h ≤ k) {
  if (b[h] < x) h = h + 1;
  else if (b[k] > x) k = k − 1;
  else {
    Swap b[h] and b[k];
    h = h + 1; k = k − 1;
  }
}
{R′}
Swap b[m] and b[k];
```

The algorithm always swaps values that are equal to x. Swapping them increases the likelihood that many occurrences of x in b will lead to segments of almost equal size.

(5) (Saddleback search)

```
i = 1; j = n;
{inv : 1 ≤ i ≤ m and 1 ≤ j ≤ n and
       x∈b[i..m, 1..j]}
while (b[i, j] ≠ x) {
  if (b[i, j] < x) i = i + 1;
  else j = j − 1;
}
```

(6) (Insertion sort) —as a procedure)

```
// Sort b[m..n] –by permuting its elements
public static void insertionSort(int []b,
                                 int m, int n);
  int i = m;
  {inv : b[m..i] is sorted}
  while (i < n) {
    i = i + 1;
    {Sort b[m..i] given b[m..i − 1] sorted. We give
     this loop without an invariant. It moves
     the value initially in b[i] toward the front
     until it reaches its final position.
     int j = i;
     while (j ≠ m && b[j − 1] > b[j]) }
       Swap b[j] and b[j − 1];
       j = j − 1;
     }
  }
```

(7) (Selection sort)

```
i = m;
while (i < n) {
  int j;
  // Store in j the pos. of the min of b[i..n];
       j = i;
       for (int k = i + 1; i ≤ n; i = i + 1) {
            if (b[k] < b[j]) j = k;
  Swap b[j] and b[i];
  i = i + 1;
}
```

(8) (Find)

```
{See exercise for a description}
int h = m; int k = n − 1;
while (h ≠ k) {
  int p;
```

```
    partition(b, h, k + 1, p);
    if (i < p)  k = p − 1;
    else if (i = p) {
      h = p; k = p;
    }
    else h = p + 1;
  }
```

1.8 Finding Loop Invariants

A complete discussion of finding invariants in beyond the scope of this course. However, a few important comments can be made.

First, postcondition R of a loop need only be true upon termination of the loop, while the invariant is true not only upon termination but before and after each iteration. Therefore, the invariant is true in more states than is the postcondition, and the invariant can be looked upon as a *generalization* of the postcondition. Thus, one good way to find (at least a first approximation to) the invariant is to generalize the postcondition, to change the postcondition so that it is true in more states.

Furthermore, the invariant has to be true just before execution of the loop, so one should generalize the postcondition into an invariant that can be easily established.

Here are three useful ways to do this.

- If the postcondition has the form (b and c) for some expressions b and c, then delete either b or c —use either c or b for the invariant.

  This method was used to find the invariant for exercise 0 (Linear search).

- In the postcondition, replace some expression by a fresh variable (and put suitable bounds on the fresh variable).

  For example, the postcondition for the loop that sums the elements of array $b[1..10]$ is

  > s is the sum of first 10 elements of b

  and we replace '10' in the postcondition by a fresh variable i to get the invariant

  > $0 \leq i \leq 10$ and
  > s is the sum of the first i elements of b.

  This invariant we can make true by executing $i, s = 0, 0$.

  The technique is used in example 1 (second version of Linear search) and example 3 (Binary search).

- When the pre- and post-conditions are given by diagrams, draw the invariant as a diagram that includes both the pre- and the post-condition as "instances" of it.

  For example, this technique was used in algorithm (5) (Partition). In Q, two segments of b are shown: one contains X and the other contains a bunch of values in any order (shown by "?"). R' has three sections: one contains X; another, values $\leq X$; and another, values $\geq X$. The final invariant P has all four sections. One gets Q from P by having the sections with values $\leq X$ and $\geq X$ empty, and one gets R' from P by having the section with mixed values (?) empty.

1.9 Mathematical Induction

One proves that a statement $P(i)$ is true for all natural numbers (nonnegative integers) i by *mathematical induction*. This consists of two steps:

- **Base case**: Prove that $P(0)$ is true.
- **Inductive case**: Assume $P(k)$ holds, where $0 \leq k$, and, using this assumption, prove that $P(k+1)$ holds.

Step (1) is sometimes replaced by the following step (1'), but one can prove that they are equivalent:

> (1') **Inductive case**: Assume $P(0), \cdots, P(k)$, where $0 \leq k$, and, using this assumption, prove that $P(k+1)$ holds.

Mathematical induction can be generalized to use any integer K (say) as the lower bound, instead of 0. To prove by mathematical induction that $P(k)$ holds for all integers $k \geq K$, where K is some integer, prove the following.

1. **Base case**: Prove that $P(K)$ is true.
2. **Inductive case**: Assume $P(k)$ holds, where $K \leq k$, and, using that assumption, prove that $P(k+1)$ holds.

Example 0. Let $P(k)$ be:

$$P(k): \ 2 + 4 + ... + 2 * k = k * (k + 1).$$

We prove that it holds for all natural numbers k.

1. **Base case.** For $k = 0$, $2 + 4 + \cdots + 2 * k$ is, by convention, 0 (it is the sum of zero integers). Therefore, the base case reduces to $0 = 0$, which is true. (You can also see easily that $2+4+\cdots+2*k = k*(k+1)$ for $k = 1$.) Hence, $P(0)$ and $P(1)$ both hold.

2. **Inductive case.** Assume $2 + 4 + \cdots + 2 * k = k * (k + 1)$. We have,

$$
\begin{array}{ll}
 & 2 + 4 + \cdots + (2 * (k + 1)) \\
= & \quad \langle \text{explicitly state the term } 2 * k \rangle \\
 & (2 + 4 + \cdots + 2 * k) + 2 * (k + 1) \\
= & \quad \langle \text{use the inductive assumption} \rangle \\
 & k * (k + 1) + 2 * (k + 1) \\
= & \quad \langle \text{arithmetic} \rangle \\
 & k^2 + 3 * k + 2 \\
= & \quad \langle \text{arithmetic} \rangle \\
 & (k + 1) * (k + 2)
\end{array}
$$

   which proves $P(k + 1)$.

Example 1. Let $P(k)$ be:

$$P(k): \ 3^k \geq 2 * k + 1.$$

We show that it holds for all natural numbers k –note that $3^0 = 1$:

1. **Base case.** For $k = 0$, $1 = 3^k = 2 * k + 1$, so $P(0)$ holds.

2. **Inductive case.** Assume $3^k \geq 2 * k + 1$. We have,

$$
\begin{array}{ll}
 & 3^{k+1} \\
= & \quad \langle \text{arithmetic} \rangle \\
 & 3^k * 3 \\
\geq & \quad \langle \text{inductive assumption} \rangle \\
 & (2 * k + 1) * 3 \\
= & \quad \langle \text{arithmetic} \rangle \\
 & 6 * k + 2 + 1 \\
= & \quad \langle \text{arithmetic} \rangle \\
 & 2 * (3 * k + 1) + 1 \\
\geq & \quad \langle \text{arithmetic} \rangle \\
 & 2 * (k + 1) + 1
\end{array}
$$

   which proves $P(k + 1)$.

Remark Note the forms of proof of the inductive case in examples 0 and 1. In each, the proof consists of a sequence of equalities and inequalities, which show that the formula on the first line equals (in the first case) or is at least (in the second case) the formula on the last line. With each equality or inequality is given a justification for it.

Mathematical induction will be used in a number of places later on in CS211. Mathematical induction can be generalized to work over other sets as well as the integers, but the generalization is beyond the scope of CS211.