Per poter testare la correttezza di un programma, si applica il famoso principio di induzione.
Ciò significa che:
-in termini matematici, se l'espressione vale nel caso [n], dovrà valere anche per [n+1]
-in termini pratici significa che, al di là dei casi base, il programma vale per ogni input e quindi è corretto perché si sa che funzionerà per i casi considerati dall'algoritmo in questione.

Esempio pratico e basico della ricorsione:

```
fun fact n =
  if n = 0
  then 1
  else n * fact (n - 1)
```

Dobbiamo provare che funziona sempre, o quantomeno funziona per ogni input.
Quindi, prima di tutto, si considerano i casi base.
In questo caso: fact(0) = 0! = 1
Essendo un programma ricorsivo, si sa che la pila della chiamate ricorsive finisce sullo stack e, come tale, il principio induttivo è applicato già.

Per il passo induttivo (n+1)
Sappiamo quindi che, se n != 0 allora sarà >0, quindi positiva.
A questo punto noi dobbiamo dimostrare che n(n+1)=n! per ogni n.
Il primo caso sarà n=1, quindi darà 0.
Poi n=2, che darà 2, n=3, che darà 6, ecc.
Tutto dipende dal valore di n, fintanto che sarà maggiore di 0 la funzione continuerà a chiamarsi ricorsivamente.

Se l'algoritmo sopra fosse stato iterativo, si considera l'invariante.
Nel qual caso:

$$
\begin{aligned}
& i = 1; \\
& factorial = 1; \\
p :& (factorial = i!) \wedge (i \leq n) \\
& \textbf{while } (i < n) \\
S :& \quad i++; \\
& \quad factorial = factorial * i;
\end{aligned}
$$

L'invariante è dato da i!=factorial && 0<=i<=n
Quindi deve essere vero nel caso 0, dove appunto l'indice i vale 0, cosa che è verificata e si piò subito vedere. SI può vedere che il fattoriale aumenta esponenzialmente all'aumentare dell'indice, pertanto la varianza del ciclo è data da entrambe le variabili e l'invarianza nel mantenimento della condizione di ciclo.

Altre cose da dire sono:

1) la presenza di precondizioni e postcondizioni di un programma, cioè dire cosa c'è prima che un programma vada in esecuzione (es. pratico, se ho un programma con due array, posso dire che la precondizione è che siano formati da n elementi non vuoti) e dopo aver eseguito (ad esempio, se avessi un algoritmo di ordinamento, assumo come postcondizione che l'array di partenza sia ordinato).
2) la presenza di cicli, ciascuno dei quali deve avere un proprio invariante (*loop invariant*), una condizione che deve valere prima di entrare nel ciclo, ad ogni sua iterazione e finire con il caso base (condizione del while o del ciclo for). Ciascun ciclo ha il suo invariante e la dimostrazione parte dal ciclo più esterno in assoluto, in presenza di cicli annidatii.

3) la presenza di if (condizionali), che comportano una variazione o meno del comportamento del programma a seconda dei dati in input.

A livello pratico, significa che l'algoritmo risolve singolarmente dei sottoproblemi e poi, a seconda di ciò che capita, si risolve e si considera una soluzione. Nei link presenti sotto si riportano alcune notazioni logiche che teoricamente dovrebbero aiutare.
Io consiglio una logica pratica e funzionale:
-partire dai casi base, che sono i primi pezzi con cui un algoritmo si realizza
-poi verificare i cicli, i quali avranno un invariante e magari alcuni if.

Esempio pratico:

Given the algorithm for the following problem:
Input:

- The starting address of a matrix of integer A of size $n \times n$
- The starting address of a matrix of integer B of size $n \times n$
- A function $matrix(16x16) : getBlock(address : X, int : i, int : j)$ which returns a sub-matrix (a block) of the matrix starting at address X, of size 16×16 whose first element is at position i, j

Ouput:

- An integer c, the sum of the diagonal elements of the product of A and B

Exercise: Prove it computes $tr(A.B)$

Algoritmo:

```
int matrix(int(*A)[16], int(*B)[16], int s_a, int s_b){
        int sum=0;
        for(int i=s_a; i<16; i++){
                for(int j=s_b; j<16; j++){
                        sum+=A[i]*B[j];
                }
        }
        return sum;
}
```

Per provarne la correttezza:
- caso base, s_a o s_b sono 16, dimensione massima, e non si fa nulla
- caso induttivo, abbiamo due cicli innestati, ciascuna con la precondizione di avere s_a/s_b diversa da 0, e come invariante 0<=i<=16 per quello esterno e 0<=b<=16 per quello più interno.
All'interno del ciclo non si hanno condizionali, ma si hanno solo assegnazioni, in particolare il prodotto della somma delle diagonali della matrice. Questa continua fintanto che valgono gli invarianti descritti.
Alla fine si ritorna la variabile sum che qualifica l'operazione descritta.

Una cosa interessante è distinguere la ricorsione dall'iterazione.
In particolare:

- l'iterazione ragiona ad invarianti e casi base che sono dati dalla condizione di stop dei cicli

1. **Initialization:** The invariant holds prior to the first loop iteration.
2. **Maintenance:** If the invariant holds at the beginning of an arbitrary loop iteration, then it must also hold at the end of that iteration.
3. **Termination:** The loop always terminates.
4. **Correctness:** Whenever the loop invariant and the loop exit condition both hold, then P must hold.

- la ricorsione invece ha già la correttezza applicata durante lo svolgimento dell'algoritmo, in particolare:

1. The *base case*. $P(0)$ is shown to be true.
2. The *induction hypothesis*. $P(n)$ is assumed to be true for some arbitrary natural number n.
3. The *induction step*. Using the induction hypothesis, $P(n+1)$ is proved.

Non è un concetto molto difficile e, al di là del matematichese, abbastanza intuitivo. Credo sia chiaro, altrimenti lascio buoni documenti e link utili alla comprensione di questo concetto.

Link utili:

https://www.cs.cornell.edu/courses/cs312/2004fa/lectures/lecture9.htm
https://www.usna.edu/Users/cs/crabbe/2004-01/SI262/correctness/correct.pdf
http://web.cs.ucla.edu/~pouchet/lectures/doc/888.11.algo.6.pdf
https://stackabuse.com/mathematical-proof-of-algorithm-correctness-and-efficiency/
https://www.youtube.com/watch?v=ndFArXAsPsc
https://people.cs.ksu.edu/~rhowell/algorithms-text/text/chapter-2.pdf