Invariant Proofs

1 Invariant Proofs

1.1 Purpose

This handout introduces a proof structure from mathematics – proof by induction – and then shows the adaptation to computer science. The result is an **invariant proof**, which proves that an algorithm produces correct results in a general case.

2 Proof By Induction

This form of proof can prove a property, but it cannot derive the property in the first place.

2.1 Definition

A proof by induction (or inductive proof) is used to prove properties in discrete domains. There are two parts to each proof: the base case, and the inductive case.

To start with, a property is expressed for a certain count, n.

To prove a base case, a small value (0, or 1 perhaps) is plugged into n and the property is proven for that specific value.

To prove an inductive case, the property is assumed to be true at an arbitrary value k, and then proven to be true for $k + 1$.

2.2 Examples

2.2.1 Triangular numbers

Property to prove: $1 + 2 + \cdots + n = \frac{n(n+1)}{2}$

Base case: When $n = 1$, the property claims $1 = \frac{1(1+1)}{2}$. $1 = 1$ is true.

Inductive case:

Assume property is true when $n = k$: $1 + 2 + \cdots + k = \frac{k(k+1)}{2}$

Show property is true when $n = k + 1$. Plug in $k + 1$ for n on the left side of the equation, and apply assumptions/algebra:

$$\begin{aligned}
1 + 2 + \cdots + k + (k+1) &= \frac{k(k+1)}{2} + (k+1) \\
&= \frac{k(k+1)}{2} + \frac{2(k+1)}{2} \\
&= \frac{k(k+1) + 2(k+1)}{2} \\
&= \frac{(k+1)(k+2)}{2} \\
&= \frac{(k+1)((k+1)+1)}{2}
\end{aligned}$$

We have shown that $1 + 2 + \cdots + k + (k+1) = \frac{(k+1)((k+1)+1)}{2}$.

Therefore, we have proven that $1 + 2 + \cdots + n = \frac{n(n+1)}{2}$ is true for $n \geq 1$ by induction.

2.2.2 Sum of Squares

Property to prove:

$$\sum_{i=1}^{n} i^2 = \frac{n(n+1)(2n+1)}{6}$$

Base case: When $n = 1$, the property is easy to prove with simple arithmetic:

$$\begin{aligned}\sum_{i=1}^{1} i^2 &= \frac{1(1+1)(2\times 1+1)}{6}\\ 1^2 &= \frac{6}{6}\end{aligned}$$

Inductive case:

Assume property is true when $n = k$:

$$\sum_{i=1}^{k} i^2 = \frac{k(k+1)(2k+1)}{6}$$

Show property is true when $n = k+1$. Plug in $k+1$ for n on the left side of the equation, and apply the assumption, followed by algebraic acrobatics:

$$\begin{aligned}\sum_{i=1}^{k+1} i^2 &= \frac{k(k+1)(2k+1)}{6} + (k+1)^2\\ &= \frac{2k^3+k^2+2k^2+k}{6} + \frac{k^2+2k+1}{6}\\ &= \frac{2k^3+9k^2+13k+6}{6}\\ &= \frac{(k+1)(k+2)(2k+3)}{6}\\ &= \frac{(k+1)((k+1)+1)(2(k+1)+1)}{6}\end{aligned}$$

We have shown that

$$\sum_{i=1}^{k+1} i^2 = \frac{(k+1)((k+1)+1)(2(k+1)+1)}{6}$$

Therefore, we have proven that $\sum_{i=1}^{n} i^2 = \frac{n(n+1)(2n+1)}{6}$ is true for $n \geq 1$ by induction.

3 Invariant Proofs

Invariant proofs are useful tools that can prove that algorithms work correctly, independent of input order or size. They adopt much of their logical proof structure from inductive proofs, especially the focus on a base and inductive case.

3.1 Definition

A property is expressed that remains true over the full run of an algorithm.

This property is supported with three sections:

- The **initialization** section proves that the property is true at the beginning of algorithm's run.
- The **maintenance** section assumes that the property is true after an arbitrary time in the algorithm's run (corresponding to some number of times through a loop or recursive calls), and the property is proven to remain true after the algorithm executes another loop/recursive call.
- The **termination** section proves that the algorithm terminates, and combines the invariant property with the termination condition to prove a more general property about the algorithm's correctness.

3.2 Purpose

Invariant proofs are presented to prove that an algorithm works correctly. Invariant properties are rarely the property that we want to prove. Instead, a proven invariant property combined with a termination condition can prove that an algorithm's result is correct.

3.3 Examples

3.3.1 Radix Sort

Property to prove: The radix sort algorithm sorts any input.

Assumptions: Input values are non-negative integers and Stable Digit Sort works correctly.

Invariant property: After sorting i columns (starting on the right), our values are sorted if we truncate them to i digits.

Initialization: When the algorithm starts, i is 0, and truncating values to digits give us 0s for each value. A list of 0s is sorted so our property holds to start with.

Maintenance: Assume that k columns have already been sorted, so the first k digits of all values are in sorted order. Show that, after running a Stable Digit Sort on column $k+1$, the first $k+1$ digits are sorted.

Running Stable Digit Sort guarantees that column $k+1$ is sorted. We can guarantee that columns $1 \dots k$ stay sorted because the Digit Sort is stable. If there is a tie between two values in column $k+1$, columns $1 \dots k$ are sorted to start with and the values in $1 \dots k$ do not swap to an incorrect ordering. Therefore columns $1 \dots k+1$ are now sorted.

Termination: The loop in radix sort runs through all d columns, so all columns are sorted. Therefore the full non-truncated input values are now in sorted order.

Like the inductive step in induction proofs, the maintenance step is the critical step to prove that an algorithm behaves correctly with any input size. The invariant property in this proof follows a frequent pattern of invariant properties: it proves something is true *for the data we have processed so far*. That ensures that the property can stay true during the entire run of the algorithm, regardless of input size.

3.3.2 Partition

Property to prove: The partition algorithm partitions `A[l..r]`.

Assumptions: $r > l$.

Invariant property: At the beginning of the for-loop, values in the range `A[l+1..i]≤p` and values in the range `A[i+1..j-1]>p`.

Initialization: Both the `A[l+1..i]` and `A[i+1..j-1]` ranges are empty, so the two parts of the invariant property are trivially true.

Maintenance: Assume that the property is true up to an index of k: `A[l+1..i]≤p`, `A[i+1..k]>p`. Show that it is true up to $k+1$ after running the for-loop one time.

If `A[k+1]>p`, no values are swapped and i is unchanged. The `>p` partition has increased in size by 1 and the `≤p` partition has not changed, so the invariant property remains true. If `A[k+1]≤p`, both i and j are incremented, which increases the `≤p` partition's size by 1. The current value is swapped into that partition, and a value `>p` (the value at i) is swapped to the `>p` partition. Therefore the invariant property stays correct in that case too.

Termination: We looped through the full `A[l+1..r]` range. Therefore `A[l+1..i]≤p` and `A[i+1..r]>p`. p is swapped with `A[i]`, which ends our algorithm with the condition that `A[l..i-1]≤p`, `A[i]=p`, `A[i+1..r]>p`. The range is correctly partitioned.