Correctness of algorithms

- An algorithm is correct if for any valid input it produces the result required by the algorithm's specification.
- For example,

static void Arrays.sort(int[] a)

is specified in the Java API as follows: *sorts the specified array of ints into ascending numerical order*.

- Arrays.sort is correct if it does exactly that for any input of type int[].

Verifying correctness of algorithms

- Ways to make sure that an algorithm is correct:
  - Testing (obvious problems with exhaustiveness)
  - Model-checking
  - Proving correctness using assertions and invariants (most of this lecture)
  - Correctness by design: construct the algorithm in the first place so that it has the desired properties (declarative programming, deriving algorithms programming algebra style)

Additional reading

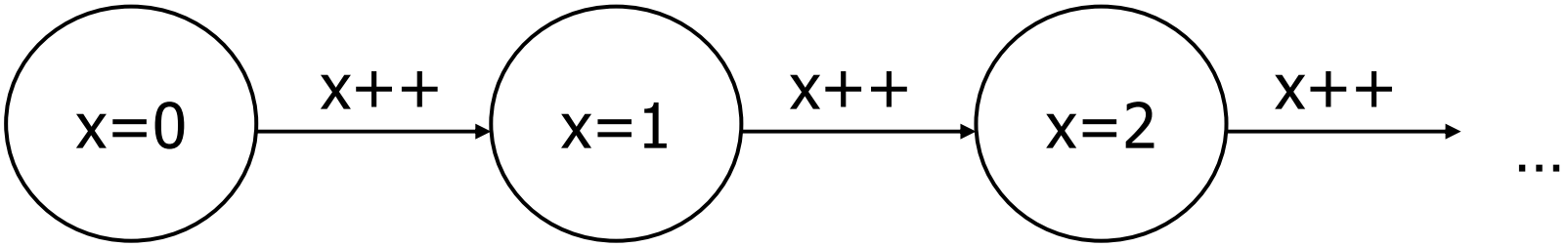
- On model-checking (and Hoare logic as well): Michael Huth and Mark Ryan, *Logic in Computer Science: Modelling and Reasoning about Systems,* Cambridge University Press, 2nd edition 2004*.*
- Goodrich and Tamassia, Chapter 4.3
- Frank M. Carrano, Janet J. Prichard. *Data abstraction and problem solving with Java.* Addison Wesley Longman, 2001. Chapter 1, Problem solving and software engineering (on verification).
- Duane A. Bailey. *Data structures in Java for the principled programmer*. McGraw-Hill 1999. Chapter 2, Comments, conditions and assertions (pre- and postconditions).
- Roland Backhouse. *Program Construction : Calculating Implementations from Specifications.* John Wiley & Sons 2003*.*

Model-checking

- Model-checking is a widely used technique for:
  - Hardware verification (chip design)
  - Verification of concurrent processes, e.g. mutual exclusion protocols, security protocols
- Increasingly used for program verification as well.
- In 1998, SPIN model-checker was used to verify plan execution module in NASA's DEEP SPACE 1 mission and discovered five previously unknown concurrency errors.

Model-checking

- Represent the program to be verified as a state transition system (states are values of variables, transitions are atomic actions of the program)
- For example, while(true) {x++;} has the following transition system:



Model-checking

- Exhaustively search the transition system for `error states', where specified property does not hold.
- Problems:
  - State explosion (but methods allowing to deal with more that 10^{20} states were developed in the 1990s; modern model-checkers can cope with a million of state variables)
  - Infinite state systems (but there are ways of representing some of them finitely)

Proving correctness using assertions

- Formulate precisely the property which has to hold
- Formulate relevant properties for smaller parts of an algorithm : program assertions
- Use assertions and additional axioms to derive the property

Assertions

- Assertion: claim about values of program variables before or after a statement or a group of statements is executed

  Typical assertions:

- Precondition (usually of a method): what we expect to hold before the method is executed.
- Postcondition: what holds after the method is executed.

Hoare triples

- {P} S {Q}: P precondition, S statements, Q postcondition.
- Meaning: provided P holds before S is executed, then after S is executed, Q holds.
- for example:

{x < 10} **`x = x + 20`** {x < 30}

{x < y} **`while (x < y) x++`** {x=y} (x,y integers)

- For a small programming language, can provide axioms for every construct in the language and derive postconditions using axioms

Assignment axiom

- If the only programming construct was assignment, here is an axiom to verify all programs:

{Q(e substituted for x)} **`x = e`** {Q}

- For example, if want to prove

{x < 10} **`x = x + 20`** {x < 30} then the assignment axiom gives

{x+20 < 30} **`x = x + 20`** {x < 30} and from extra knowledge about math etc we derive that {x+20 < 30} is equivalent to {x < 10}.

Example

- In the programs we usually write there are lots of constructs and they also use other people's code.
- Less formal approach (but good practice): write pre- and postconditions for significant chunks of code/methods.
- Example: code in Bailey's book.

Proving correctness

To prove that an algorithm is correct:

- Determine preconditions and postconditions for the whole algorithm.
- Cascade statement assertions together, so that postconditions for one provide preconditions for the next.
- Prove correctness of individual statements.
- Hence show that executing algorithm with stated preconditions terminates and leads to stated post-conditions.

Loop Invariants

- Assertions for loops are difficult, because loops may be executed many times over, with slightly different assertions holding before and after each iteration. Focus on those assertions that remain constant between iterations.

- Known as l*oop invariants:* true before and after each iteration through a loop.

Example

```
pos_greatest = 0;
for (int j = 0; j < =i; j++) {
   if( arr[j] > arr[pos_greatest]) {
      pos_greatest = j;
   }
}
```

Invariant: **`pos_greatest`** is the index of the largest array element between 0 and j.

(More formally, for all k such that **`0 ≤ k < j`**, **`arr[k] ≤ arr[pos_greatest].)`**

Correctness of loops

To prove correctness of a while loop (or: that assertion A holds after the loop terminates):

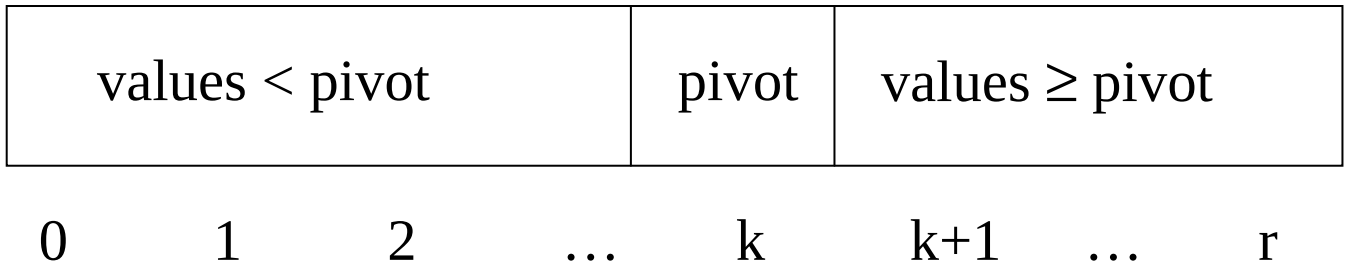
- Prove that the loop eventually terminates (by finding the *bound function* for the loop)
- Find a suitable invariant (there are infinitely many invariants for each loop, most of them useless)
- Prove that A is true after last iteration (usually by substituting the state in which the loop terminates in the invariant)

Partition algorithm

```
small = l;  // set at the left border of the
            // range
large = r; // set at the right border where the
            // pivot sits
while(small < large) {
   if (arr[small] < pivot) small++;
   else {
         large--;
         temp = arr[small];
         arr[small] = arr[large];
         arr[large] = temp; }
}
temp = arr[r];
arr[r] = arr[large];
arr[large] = temp;
return large;
```

Postcondition for the partition

```
public int partition(int[] arr, int l, int r)
// post: returns and integer k such that
//       for all indices i such that l<=i<k,
//       arr[i]<arr[k] and
//       for all indices i such that k<=i<r
//       arr[i]>=arr[k]
```

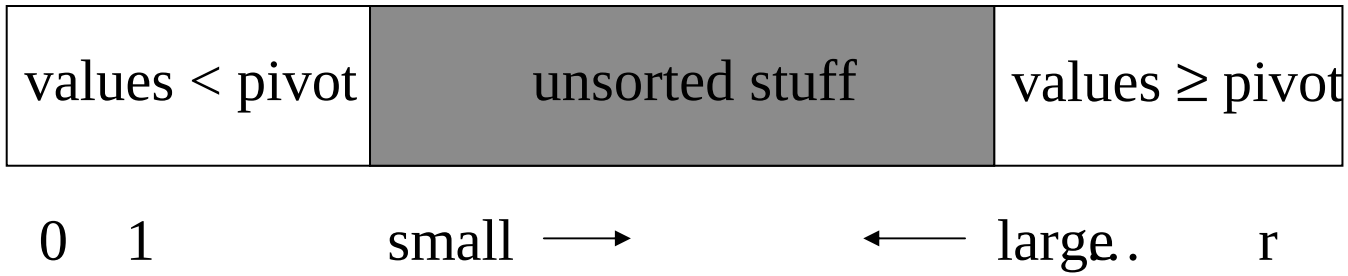


Loop invariant for partition

for all indices i,

if `l ≤ i < small`, then `arr[i] < pivot`

if `large ≤ i < r` then `arr[i] ≥ pivot`



Bound function for partition

- bound function = `large - small`
- decreases by 1 at every step; the loop terminates when it is equal to 0 (`small = large`).

After the loop…

- When **`large = small`**, the invariant:
  - if **`l ≤ i < small`**, then **`arr[i] < pivot`**
  - if **`large ≤ i < r`** then **`arr[i] ≥ pivot`**

becomes (substitute **`large`** for **`small`**):

  - if **`l ≤ i < large`**, then **`arr[i] < pivot`**
  - if **`large ≤ i < r`** then **`arr[i] ≥ pivot`**

- After the pivot is swapped, **`pivot = arr[large].`**

Partition algorithm

```
Assertion 1: l < r; pivot is arr[r]
while(small < large) {
   if (arr[small] < pivot) small++;
   else {
        large--;
        temp = arr[small];
        arr[small] = arr[large];
        arr[large] = temp;}}
Assertion 2: pivot is arr[r];
if l ≤ i < large, then arr[i] < pivot;
if large ≤ i < r then arr[i] ≥ pivot
temp = arr[r];
arr[r] = arr[large];
arr[large] = temp;
Assertion 3: arr[large] is the pivot
return large;
```

Questions for revision

Given that l < r in the partition method, which of the following are loop invariants of the while loop:

1) small < r

2) small < large

3) small ≤ large

4) for all i such that l ≤ i < small, arr[i] < pivot

5) for all j such that large ≤ j ≤ r, arr[j] ≥ pivot

6) for all j such that large < j ≤ r, arr[j] ≥ pivot