CS 130 SOFTWARE ENGINEERING

HOARE LOGIC:
WEAKEST PRECONDITION
LOOP INVARIANT

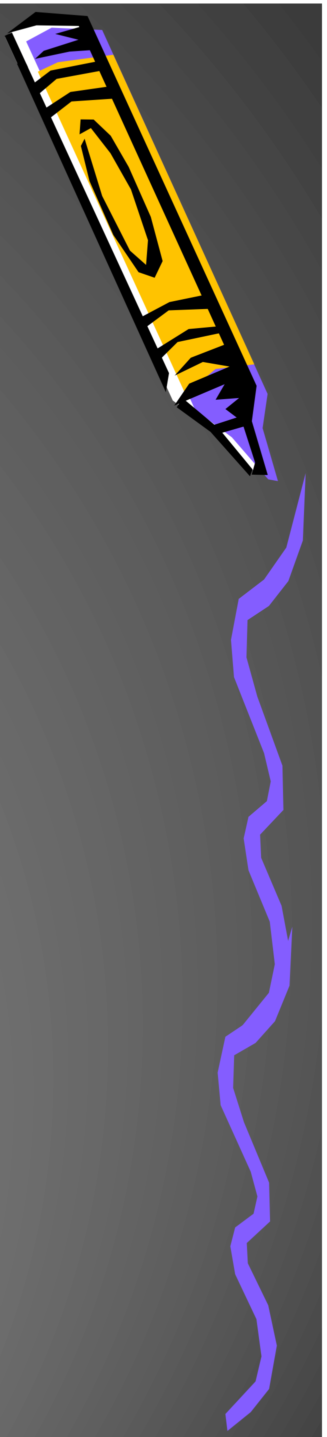
Professor Miryung Kim
UCLA Computer Science
Based on Materials from Miryung Kim

- Assignment 2 will be uploaded today
- Due next next Wednesday 11:59 PM

AGENDA

- Recap Exercise: computing weakest precondition for a loop-free program
- Why study loop invariants? We need to know how to reason about the behavior of loops.
- Which properties do hold during the execution of a loop?

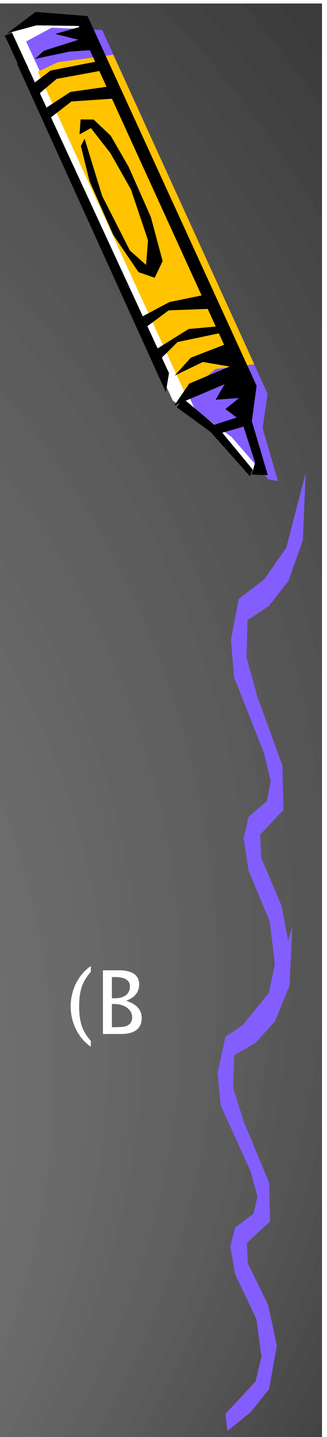
REVIEW



- {P} skip {P}
- {P[w:=E]} w:=E {P}
  - wp(w := E, R) ≡ R[w := E]
- {P ∧ B} assert B {P}
  - wp(assert P, R) ≡ P ∧ R

REVIEW

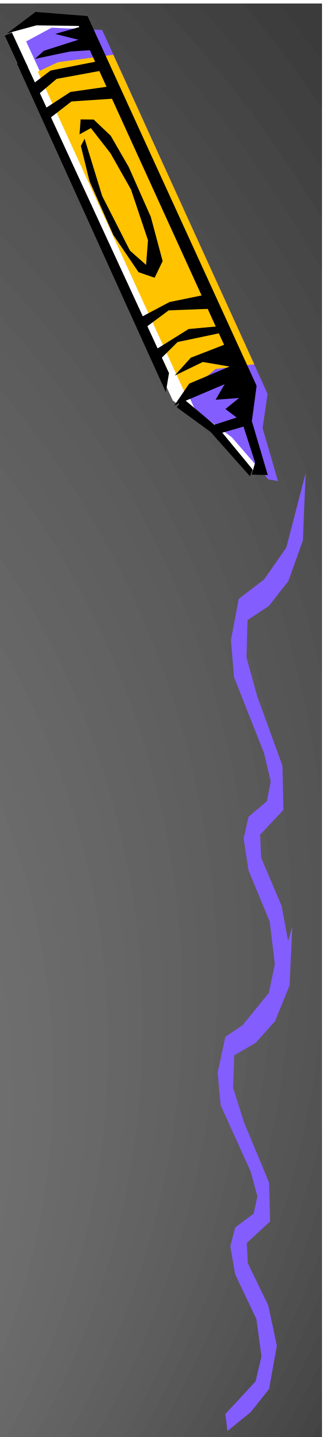
- if {P} S {Q} and {Q} T {R}, then {P} S ; T {R}
  - wp(S;T, R) ≡ wp(S, wp(T, R))
- if {P $\wedge$ B} S {R} and {P $\wedge$ $\neg$B} T {R}, then {P} if B then S else T end {R}
  - wp(if B then S else T end, R) = (B $\wedge$ wp(S, R)) $\vee$ ($\neg$B $\wedge$ wp(T, R))



THINK PAIR SHARE

During peer code reviews, under what basis should programmers approve code changes?

- Look for tests
- Conform to style guidelines
- Comments



THINK PAIR SHARE

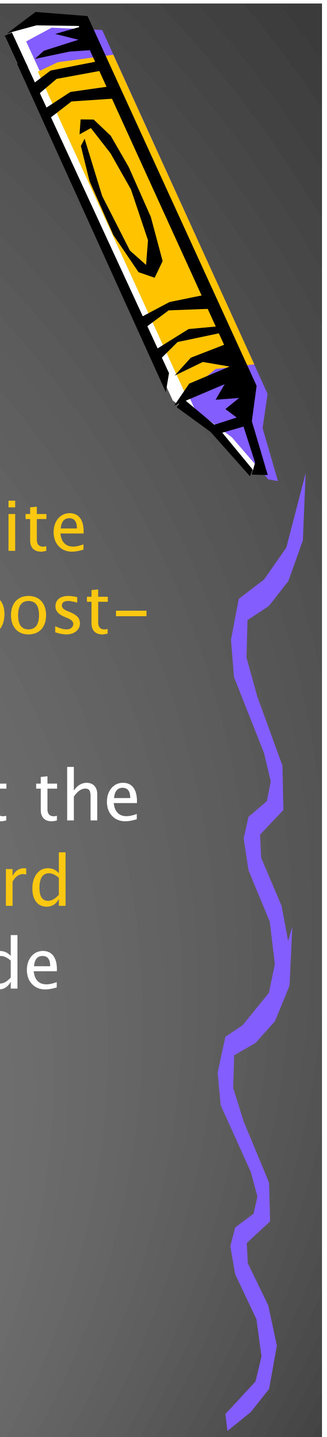
During peer code reviews, under what basis should programmers approve code changes?

- Pre-condition / post-conditions
- Comments
- Presence of tests
- Style check – indentation check
- Variable names reasonable in semantics length
- Run existing tests

THINK PAIR SHARE

- We talked about the following sequential and composition WP rule. What is the implication of this rule in practice when doing code review?
  - $wp(S;T, R) \equiv wp(S, wp(T, R))$

THINK PAIR SHARE



- $wp(S;T, R) \equiv wp(S, wp(T, R))$
  - If you authored the code, you should write comments or annotations to make the post-condition R explicit.
  - If you are debugging code, print out out the failing program state, and apply backward WP reasoning to figure out when the code fails.

EXERCISE 1.

```
public char[] foo(Object x, int z)
  if (x != null) {
     n = x.f;
  } else {
     n = z-1;
     z++;
  }
  a = new char[n];
 return a;
}
```

Suppose Alice wrote `foo`. **Which arguments need to be passed to `foo` so that it returns a non null value without throwing any `NullPointerException` and `ArrayOutOfBoundException`?**

EXERCISE 1.

which pre-condition should hold here?

```
if (x != null) {
   n = x.f;
} else {
   n = z-1;
   z++;
}
a = new char[n];
```

true

EXERCISE I.

```
if (x != null) {
    n = x.f;
} else {
    n = z-1;
    z++;
}
a = new char[n];
```

EXERCISE 1.

(x != null AND x.f >= 0) OR
(x == null AND   z-1 >= 0)

```
if (x != null) {
    n = x.f;
} else {
    n = z-1;
    z++;
}
a = new char[n];
```

x.f >= 0

z-1 >= 0

n >= 0

true

THINK PAIR SHARE

- Suppose the weakest precondition is
- (x != null AND x.f >= 0) OR
  (x == null AND z-1 >= 0).
- If you are a developer, what should you do?
- If you are a tester, which inputs should you use?

THINK PAIR SHARE

- Suppose the weakest precondition is
- (x != null AND x.f >= 0) OR
  (x == null AND z-1 >= 0).
- If you are a developer, what should you do?
  - Write assertions in the beginning of the code.
  - Update JAVADOC to write the WP as a precondition contract

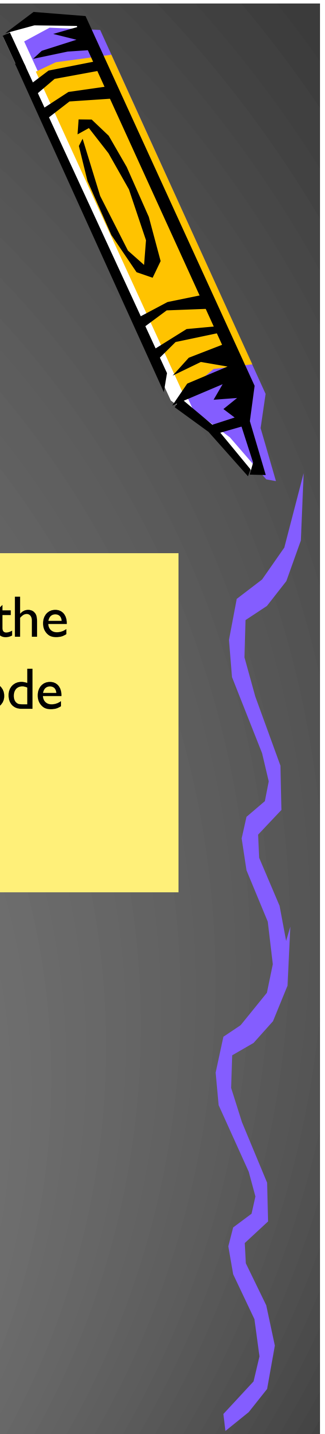
THINK PAIR SHARE: TESTING

- Suppose the weakest precondition is
- (x != null AND x.f >= 0) OR
  (x == null AND   z-1 >= 0).
- If you are a tester, which inputs should you create?
  - T1. X=new Obj(); x.setF(1); => PASS
  - T2. X=new Obj(); x.setF(-1); => FAIL
  - T3. X == null; z=1 => PASS
  - T4. X == null; z=0 => FAIL ...

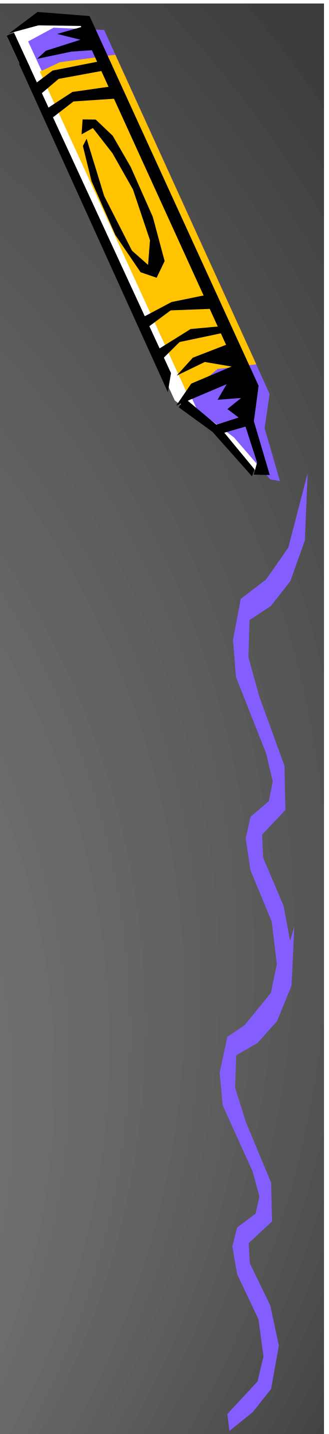
EXERCISE 2.

```
if (x!= null) {
 n =x.f;
} else {
 n = z+1;
 z = 2*z+1;
   }
a = new char[n-2];
c = a[z];
```

Suppose Tom wrote this code. What is the weakest pre-condition such that this code terminates **without throwing any exceptions**?



```
if (x!= null) {
 n =x.f;
} else {
 n = z+1;
 z = 2*z+1;
   }
 a = new char[n-2]; //
 c = a[z];
```



```
 {x!=null AND wp(n:=x.f, n>=2, n>z+2, z>=0} OR
{x==null AND wp (n:=z+1,z:=2z+1, n>=2, n>z+2, z>=0)}
 ==={x!=null AND x.f>=2, x.f>z+2, z>=0} OR {x==null
AND wp (n:=z+1, n>=2, n>2z+1+2, 2z+1>=0)
 ==={x!=null AND x.f>=2, x.f>z+2, z>=0} OR {x==null
AND z+1>=2, z+1>2z+3, 2z+1>=0)}
 ==={x!=null AND x.f>=2, x.f>z+2, z>=0} OR {x==null
AND z>=1, 0>z+2, 2z+1>=0)
 ==={x!=null AND x.f>=2, x.f>z+2, z>=0} OR {x==null
AND FALSE, 2z+1>=0)}
 if (x!= null) {
   n =x.f;
 } else {
   n = z+1;
   z = 2*z+1;
       }
 wp(a:=new char[n-2], z<a.length, z>=0)
 === {n-2>=0, z<n-2, z>=0}
 === {n>=2, n>z+2, z>=0}
  a = new char[n-2]; //
 z>=0 AND z<a.length
  c = a[z];
 TRUE
```

THINK-PAIR-SHARE

- The WP is (x!=null AND x.f>=2 AND x.f>z+2 AND z>=0) or (FALSE)
- If you are a developer, what would you do based on the weakest precondition?

THINK-PAIR-SHARE

- The WP is (x!=null AND x.f>=2 AND x.f>z+2 AND z>=0) or (FALSE)
- If you are a developer, what would you do based on the weakest precondition?
  - Report bug with an input x=null;
  - Remove the else branch
  - Add an "assert" as a guard

EXERCISE 3. ARRAY SUM

```
k := 0; s := 0;
while k ≠ N do
    s:=s+a[k] ; k:=k+1
end
```

Suppose that Tom wrote the above code to compute the sum of integer elements in the array. **Is this code correct?**

REASONING ABOUT LOOPS

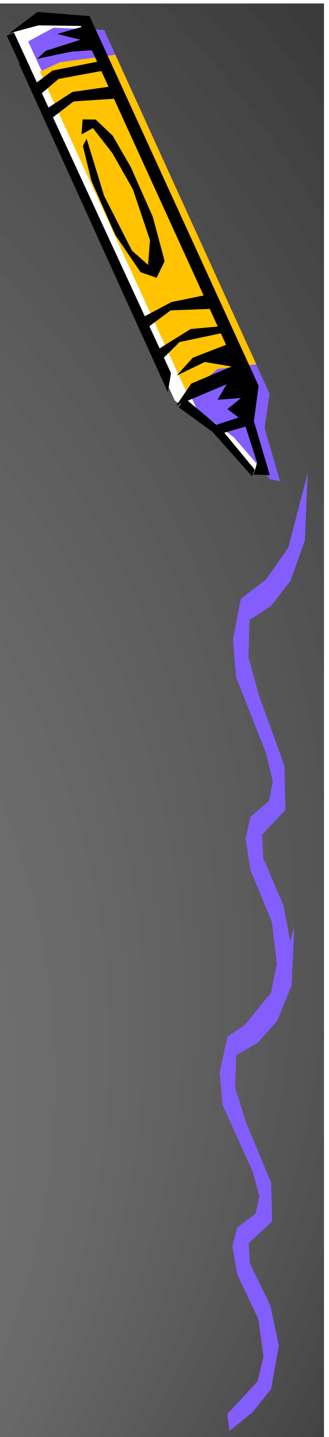
{P} while B do S end {Q}

invariant J and variant function vf such that:

- (1) invariant initially: $P \Rightarrow J$
- (2) invariant maintained: $\{J \wedge B\}\ S\ \{J\}$
- (3) invariant sufficient: $J \wedge \neg B \Rightarrow Q$
- (4) vf bounded: $J \wedge B \Rightarrow 0 \leqq vf$
- (5) vf decreases: $\{J \wedge B \wedge vf = VF\}\ S\ \{vf < VF\}$

THINK-PAIR-SHARE

- Why do we care about loop invariants and what are the implications?



THINK-PAIR-SHARE

- Why do we care to learn “loop invariants” and what are the implications?
  - You don’t have to write proofs for a loop by hand. => Software verification tools automate the proof if you write pre/post conditions with invariants.
  - Writing contract is a very good practice!
  - Partial loop unrolling is still useful for finding bugs.

APPLICATION OF HOARE LOGIC/ LOOP INVARIANT

- It's good to actively think about the semantics of a loop.
- To check correctness about a loop, you can first check whether an invariant holds in the beginning after the initialization.
- You can check whether the invariant is maintained in the iteration.
- You can check whether the invariant and the exit condition implies a post condition.

PEER REVIEW SCENARIO

```
k := 0; s := 0;
while k ≠ N do
   s:=s+a[k] ; k:=k+1
end
```

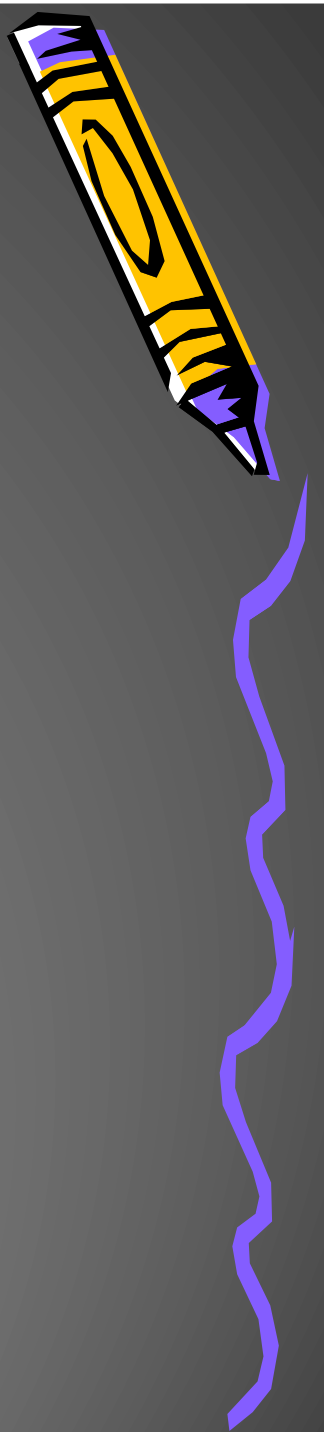
Q1. Suppose that Tom wrote the above code to compute the sum of integer elements in the array. For Tom to expect that the following post condition holds, given the precondition of {0<=N}, in other words, s is the sum of all elements of array a[], {s = (Σi | 0≦i<N · a[i]), **which invariant J must hold during the execution of the loop**?

PEER REVIEW SCENARIO

```
k := 0; s := 0;
while k ≠ N do
    s:=s+a[k] ; k:=k+1
end
```

Q2. **Tom's team lead wants a proof that the above loop behaves correctly.** In other words, provide three proofs that (1) the loop invariant holds initially P=>J, (2) the loop invariant is maintained {J^B} S {J}, and (3) the invariant is sufficient, J^not(B) =>Q where P is the precondition and Q is the post-condition.

EXERCISE.ARRAY SUM



P{0 ≦ N}

```
k := 0; s := 0;
while k ≠ N do
        s:=s+a[k] ; k:=k+1
end
```

Q: {s = (Σi | 0 ≦ i<N ・ a[i])}

J: 0<=k<=N AND s = (Σi | o ≦ i<k ・ a[i])}

EXERCISE.ARRAY SUM

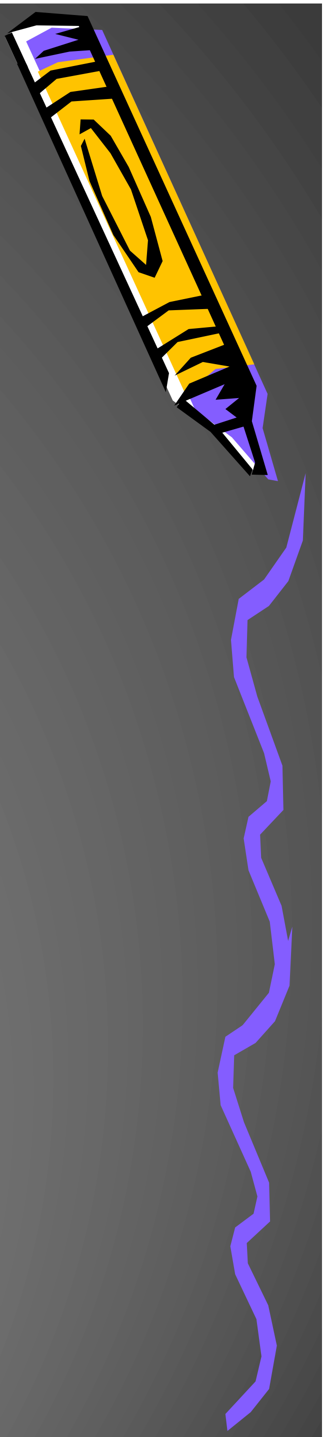
Guess Invariant J and a bounded function vf.

J: $s = (\Sigma i \mid 0 \leqq i < k \cdot a[i])$

$\wedge\ 0 \leqq k \leqq N$

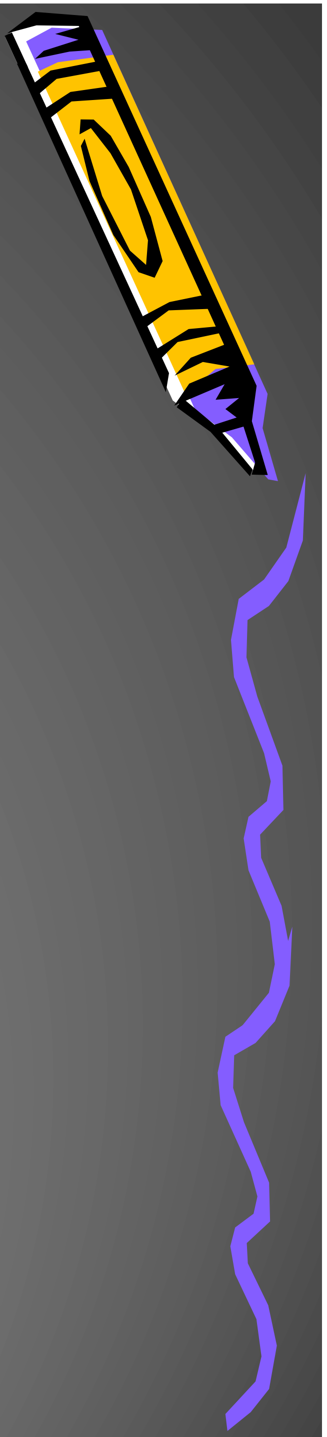
vf: $N-k$

We then show how the five conditions hold for J and vf.



THINK-PAIR-SHARE

- How do you guess a loop invariant?
- How do you guess a variant function?
- Can we automatically find loop invariants?
- Can we infer pre and post conditions?



TIPS FOR LOOP REASONING

- How do you guess a loop invariant?
  - Generally by modifying a post condition and modifying a guard.
- How do you guess a variant function?
  - Generally by coming up with a function N-k if k increases by 1.
- Can we automatically find loop invariants?
  - Yes, it's an active research area to find a loop invariant. Mostly they generate candidates and verify.
- Can we infer pre and post conditions?
  - Yes, There is also work that infers pre/post conditions from tests

- Caveats:
  - There are more than one loop invariants.
  - In the absence of a human-provided meaningful post-condition, the loop invariant subsequently is often not meaningful as well.

EXERCISE.ARRAY SUM

—(1) INVARIANT INITIALLY: {P} CODE $\Rightarrow$ {J}

P:$\{0 \leqq N\}$

$\{0 = (\Sigma i \mid o \leqq i < 0 \cdot a[i]) \wedge o \leqq 0 \leqq N\}$

k := 0;

$\{0 = (\Sigma i \mid o \leqq i < k \cdot a[i]) \wedge o \leqq k \leqq N\}$

s := 0;

J:$\{s = (\Sigma i \mid 0 \leqq i < k \cdot a[i]) \wedge 0 \leqq k \leqq N\}$

Proving P=>J

$\{0 \leqq N\}$=> $\{0 \leqq N\}$ (true because A=>A)

{P} code $\Rightarrow$ {J} iff {P} => wp(code, J}

EXERCISE.ARRAY SUM

—(1) INVARIANT INITIALLY: $P \Rightarrow J$

$\{0 \leqq N\}$

$\{0 = (\Sigma i \mid 0 \leqq i < 0 \cdot a[i]) \wedge 0 \leqq 0 \leqq N\}$

k := 0;

$\{0 = (\Sigma i \mid 0 \leqq i < k \cdot a[i]) \wedge 0 \leqq k \leqq N\}$

s := 0;

$\{s = (\Sigma i \mid 0 \leqq i < k \cdot a[i]) \wedge 0 \leqq k \leqq N\}$

EXERCISE.ARRAY SUM

—(2) INVARIANT MAINTAINED: $\{J \wedge B\}\ S\ \{J\}$

J AND B==={s = (Σi | 0≦i<k · a[i]) ∧ 0≦k<=N AND k!=N}

$\{s = (\Sigma i \mid 0 \leqq i<k \cdot a[i]) \wedge 0 \leqq k+1 \leqq N\}$

s := s + a[k];

$\{s = (\Sigma i \mid 0 \leqq i<k \cdot a[i])+a[k] \wedge 0 \leqq k+1 \leqq N\}$

$\{s = (\Sigma i \mid 0 \leqq i<k+1 \cdot a[i]) \wedge 0 \leqq k+1 \leqq N\}$

k := k+1;

J: $\{s = (\Sigma i \mid 0 \leqq i<k \cdot a[i]) \wedge 0 \leqq k \leqq N\}$

J: $\{s = (\Sigma i \mid 0 \leqq i<k \cdot a[i]) \wedge 0 \leqq k \leqq N\}$

To show $\{J \wedge B\}\ S\ \{J\}$ is to show $\{J \wedge B\} \Rightarrow wp\,(S,J)$ since $\{P\}S\{Q\}$ === $P \Rightarrow wp$

$\{s = (\Sigma i \mid 0 \leqq i<k \cdot a[i]) \wedge 0 \leqq k<N\}$ => $\{s = (\Sigma i \mid 0 \leqq i<k \cdot a[i]) \wedge 0 \leqq k+1 \leqq N\}$

EXERCISE.ARRAY SUM

—(2) INVARIANT MAINTAINED: $\{J \wedge B\}\ S\ \{J\}$

$\{s = (\Sigma i \mid 0 \leqq i<k \cdot a[i]) \wedge 0 \leqq k \leqq N \wedge k \neq N\}$

$\{s+a[k] = (\Sigma i \mid 0 \leqq i<k \cdot a[i])+a[k] \wedge 0 \leqq k<N\}$

s := s + a[k];

$\{s = (\Sigma i \mid 0 \leqq i<k \cdot a[i])+a[k] \wedge 0 \leqq k<N\}$

$\{\boldsymbol{s = (\Sigma i \mid 0 \leqq i<k+1 \cdot a[i])} \wedge 0 \leqq k+1 \leqq N\}$

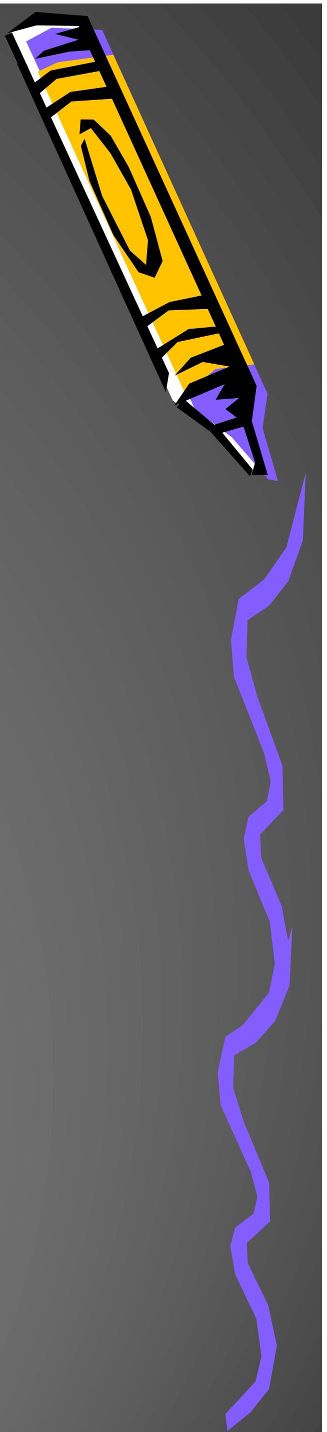
k := k+1;

$\{s = (\Sigma i \mid 0 \leqq i<k \cdot a[i]) \wedge 0 \leqq k \leqq N\}$

$s = (\Sigma i \mid 0 \leqq i<k+1 \cdot a[i])$

$= a[0] + a[1] \ldots a[k-1] + a[k]$

= sum 0<=i<k a[i]+a[k]



EXERCISE.ARRAY SUM

—(3) INVARIANT SUFFICIENT: $J \wedge \neg B \Rightarrow Q$

JAND NOT B: $\{s = (\Sigma i \mid 0 \leqq i < k \cdot a[i]) \wedge 0 \leqq k \leqq N \wedge \neg(k \neq N)\} \Rightarrow$

Q: $\{s = (\Sigma i \mid 0 \leqq i < N \cdot a[i])\}$

JAND NOT B is simplified as

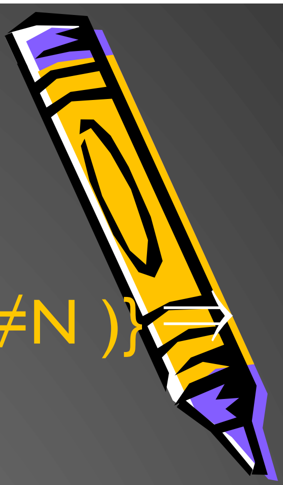
$\{s = (\Sigma i \mid o \leqq i < k \cdot a[i]) \wedge o \leqq k \leqq N \wedge k=N\}$

$== \{s = (\Sigma i \mid o \leqq i < k \cdot a[i]) \wedge k=N\}$

$== \{s = (\Sigma i \mid o \leqq i < N \cdot a[i]) \wedge k=N\}$

$\{s = (\Sigma i \mid o \leqq i < N \cdot a[i]) \wedge k=N\}$ => $\{s = (\Sigma i \mid o \leqq i < N \cdot a[i])\}$ (true!)

why A AND B => A



EXERCISE.ARRAY SUM

—(3) INVARIANT SUFFICIENT: $J \wedge \neg B \Rightarrow Q$

J AND NOT (B)

$===\{s = (\Sigma i \mid 0 \leqq i<k \cdot a[i]) \wedge 0 \leqq k \leqq N \wedge \neg(k \neq N)\} \Rightarrow$

$===\{s = (\Sigma i \mid o \leqq i<k \cdot a[i]) \wedge k==N)\}$

$===\{s = (\Sigma i \mid o \leqq i<N \cdot a[i]) \wedge k==N)\}$

Q: $\{s = (\Sigma i \mid 0 \leqq i<N \cdot a[i])\}$

J AND NOT B => Q

A AND B=> A

A AND B => B

Q is the post condition that you desire.

J ^ NOT B == the condition that holds when you exit the loop

LOOP TERMINATION VERIFICATION CONDITIONS

- vf bounded: J ^ B => (o <= vf)
- vf decreases: {J ^ B ^ vf = VF} S {vf < VF}

- First, Just like a loop invariant, vf is a piece of puzzle that you need to complete an inductive proof. So its role is similar to an inductive case where you have to guess first and show that your choice of vf satisfies the above two conditions.

LOOP TERMINATION VERIFICATION CONDITIONS

- vf is a monotonically decreasing function that converges towards 0. When vf becomes 0, that's the time that a loop terminates.
- Second, vf bounded: J ^ B => (o <= vf). This means that while the loop is running (meaning B is true and J is also true), the variant function that you picked has a positive value.

LOOP TERMINATION VERIFICATION CONDITIONS

- Third, vf decreases: {J ^ B ^ vf = VF} S {vf < VF} // This means that while a loop is iterating, vf is monotonically decreasing. Think about is vf's value is equal to VF before the execution of loop's body, but it now becomes less than VF, meaning that it is monotonically decreasing.

PEER REVIEW SCENARIO

```
k := 0; s := 0;
while k ≠ N do
    s:=s+a[k] ; k:=k+1
end
```

Q3. During a peer code review meeting, **Tom's manager asks Tom, whether the following code will always terminate without throwing any exceptions.** Find a variant function vf, and show that vf is bounded and vf decreases.

- We are going to first guess vf as N-k. (N-k is a monotonically decreasing function as k is incremented during loop's execution and it becomes 0 when k becomes N.) Right?

EXERCISE.ARRAY SUM

—(4) VF BOUNDED: $J \wedge B \Rightarrow 0 \leq vf$

vf is N-k

$\{s = (\Sigma i \mid 0 \leqq i<k \cdot a[i]) \wedge 0 \leqq k<N\} \Rightarrow$

$\{k \leqq N\}$

The above statement is true.

EXERCISE. ARRAY SUM

—(5) VF DECREASES: $\{J \wedge B \wedge vf=VF\}$ S $\{vf<VF\}$

$\{s = (\Sigma i \mid 0 \leqq i<k \cdot a[i]) \wedge 0 \leqq k \leqq N \wedge k \neq N$
$\wedge\ N-k=VF\}$

[redacted]

s := s + a[k];

[redacted]

[redacted]

k := k+1;

$\{N-k<VF\}$

EXERCISE.ARRAY SUM

—(5) VF DECREASES: $\{J \land B \land vf=VF\}\ S\ \{vf<VF\}$

$\{s = (\Sigma i \mid 0 \leqq i<k \cdot a[i]) \land 0 \leqq k \leqq N \land k \neq N$
$\land\ N-k=VF\}$

$\{N-k-1<VF\}$

$s := s + a[k];$

$\{N-k-1<VF\}$

$\{N-(k+1)<VF\}$

$k := k+1;$

$\{N-k<VF\}$

THINK PAIR SHARE

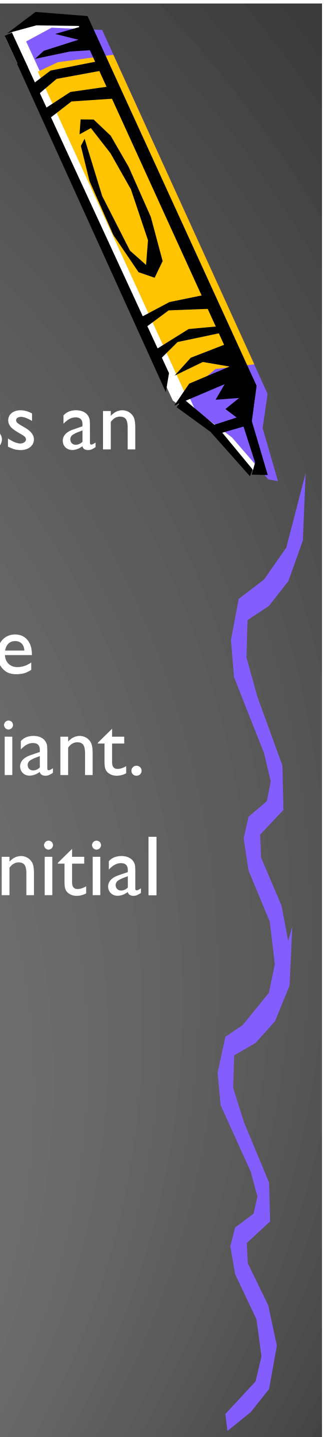
- It's time consuming and hard. Why am I learning this?
  - It's a good practice to think about what holds during a loop execution, and check whether that invariant is satisfied in the beginning and the end.
  - If you guess a loop invariant. It's good habit to write assertions during development.

RECAP

- We studied how to find errors using Weakest Precondition reasoning.
- We studied how to prove the correctness of a loop.

EXERCISE: COMPUTING CUBES

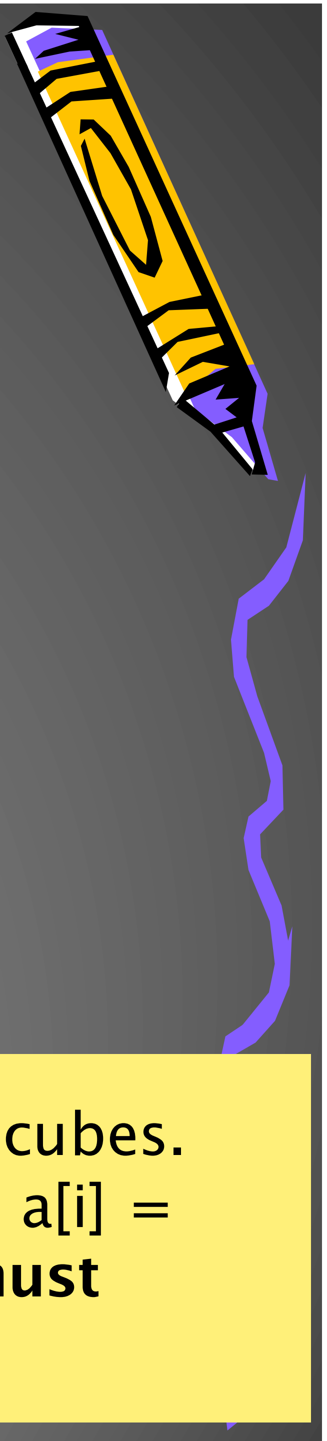
- This example shows a case where you guess an invariant that is not strong enough.
- Therefore you are going to "strengthen" the invariant by adding a new term to the invariant.
- Basically this shows how you can guess an initial invariant and refine it gradually.



EXERCISE: COMPUTING CUBES

```
k := 0;  r := 0;  s := 1;  t := 6;
while k≠N do
    a[k] := r;
    r := r + s;
    s := s + t;
    t := t + 6;
    k := k + 1
end
```

Q1. Suppose that Sheryl wrote the above code to compute cubes. For Sheryl to expect the post-condition of $\{(\forall i \mid 0 \leqq i < N \cdot a[i] = i^3)\}$ given the pre-condition of $\{0 \leqq N\}$, **which invariant must hold during the execution of the while loop?**

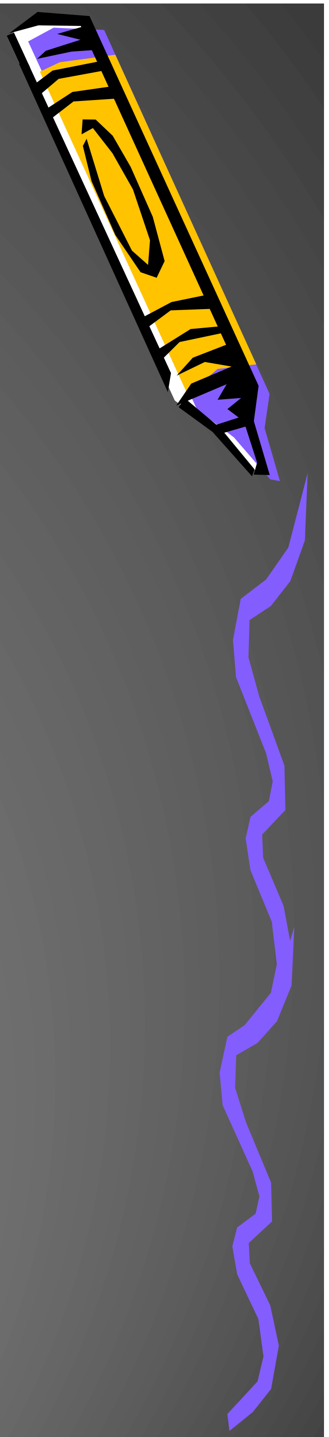


EXERCISE: COMPUTING CUBES

```
k := 0;  r := 0;  s := 1;  t := 6;
while k≠N do
    a[k] := r;
    r := r + s;
    s := s + t;
    t := t + 6;
    k := k + 1
end
```

Q2. **Suppose that Sheryl's manager wants a proof that the following loop will always terminate** when the precondition {0≦N} is satisfied. What is the variant function vf and what are the corresponding proofs that vf is bounded and vf decreases?

EXERCISE: COMPUTING CUBES



```
{$0 \leqq N$}
k := 0; r := 0; s := 1; t := 6;
while $k \neq N$ do
    a[k] := r;
    r := r + s;
    s := s + t;
    t := t + 6;
    k := k + 1
end
{$(\forall i \mid 0 \leqq i < N \cdot a[i] = i^3)$}
```

COMPUTING CUBES —GUESSING THE INVARIANT

- From the postcondition

  $(\forall i \mid 0 \leq i < N \cdot a[i] = i^3)$

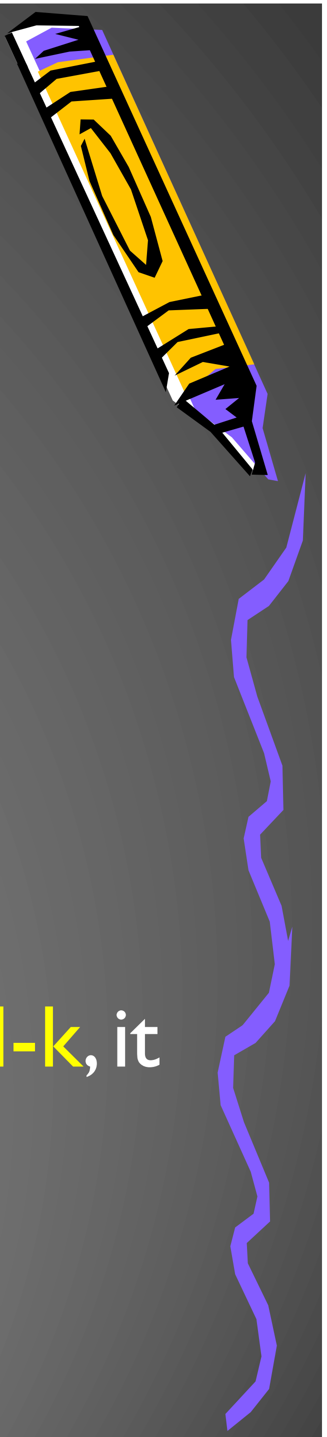
  and the negation of the guard

  $k = N$

  guess the invariant

  $(\forall i \mid 0 \leq i < k \cdot a[i] = i^3) \wedge 0 \leq k \leq N$

- From this invariant and variant function N-k, it follows that the loop terminates



COMPUTING CUBES —MAINTAINING THE INVARIANT: {J^B}S{J} === {J^B} => WP(S,J)

```
while k≠N do
    {(∀i | 0≦i<k · a[i] = i³) ∧ 0≦k≦N ∧ k≠N} (left side)
    {(∀i | 0≦i<k · a[i] = i³) ∧ r=k³ ∧ 0≦k<N} ) (right side)
    a[k] := r;
    r := r + s;
    s := s + t;
    t := t + 6;
    {(∀i | 0≦i<k · a[i] = i³) ∧ a[k]=k³ ∧ 0≦k<N}
    {(∀i | 0≦i<k+1 · a[i] = i³) ∧ 0≦k+1≦N}
    k := k + 1
    initial candiate J: {(∀i | 0≦i<k · a[i] = i³) ∧ 0≦k≦N}
end
```

Add this to the invariant, and then try to prove that it is maintained

COMPUTING CUBES —MAINTAINING THE INVARIANT

```
while $k \neq N$ do
    $\{r = k^3 \wedge \ldots\}$
    $\{r + s = k^3 + 3*k^2 + 3*k + 1\}$
    a[k] := r;
    r := r + s;
    s := s + t;
    t := t + 6;
    $\{r = k^3 + 3*k^2 + 3*k + 1\}$
    $\{r = (k+1)^3\}$
    k := k + 1
    J' that is additional to the original J $\{r = k^3\}$
end
```

Add
$s = 3*k^2 + 3*k + 1$
to the invariant, and then try to prove that it is maintained

COMPUTING CUBES —MAINTAINING THE INVARIANT

```
while k≠N do
    {$s = 3*k^2 + 3*k + 1 \wedge \ldots$} (left)
    {$s + t = 3*k^2 + 6*k + 3 + 3*k + 3 + 1$} (right hand)
    a[k] := r;
    r := r + s;
    s := s + t;
    t := t + 6;
    {$s = 3*k^2 + 6*k + 3 + 3*k + 3 + 1$}
    {$s = 3*(k+1)^2 + 3*(k+1) + 1$}
    k := k + 1
    {$s = 3*k^2 + 3*k + 1$}
end
```

Add
$t = 6*k + 6$
to the invariant, and then try to prove that it is maintained

COMPUTING CUBES —MAINTAINING THE INVARIANT

```
while $k \neq N$ do
    $\{t = 6*k + 6 \wedge \ldots\}$ left
    $\{t + 6 = 6*k + 6 + 6\}$ (right)
    a[k] := r;
    r := r + s;
    s := s + t;
    t := t + 6;
    $\{t = 6*k + 6 + 6\}$
    $\{t = 6*(k+1) + 6\}$
    k := k + 1
    $\{t = 6*k + 6\}$
end
```

COMPUTING CUBES —ESTABLISHING THE INVARIANT

$\{0 \leqq N\}$ (left)

$\{(\forall i \mid 0 \leqq i < 0 \cdot a[i] = i^3) \wedge \quad 0 \leqq 0 \leqq N \wedge$

$\quad 0 = 0^3$ (TRUE) $\wedge$

$\quad 1 = 3*0^2 + 3*0 + 1$ (TRUE) $\wedge$

$\quad 6 = 6*0 + 6$ (TRUE)$\}$ (right)

k := 0; r := 0; s := 1; t := 6;

J: $\{(\forall i \mid 0 \leqq i < k \cdot a[i] = i^3) \wedge 0 \leqq k \leqq N \wedge$

$\quad r = k^3 \wedge$

$\quad s = 3*k^2 + 3*k + 1 \wedge$

$\quad t = 6*k + 6\}$

- Okay, now we found J that satisfies {J^B}S{J}.
- Now we need first to show that P=>J
- Then we show that {J AND NOT B} => Q

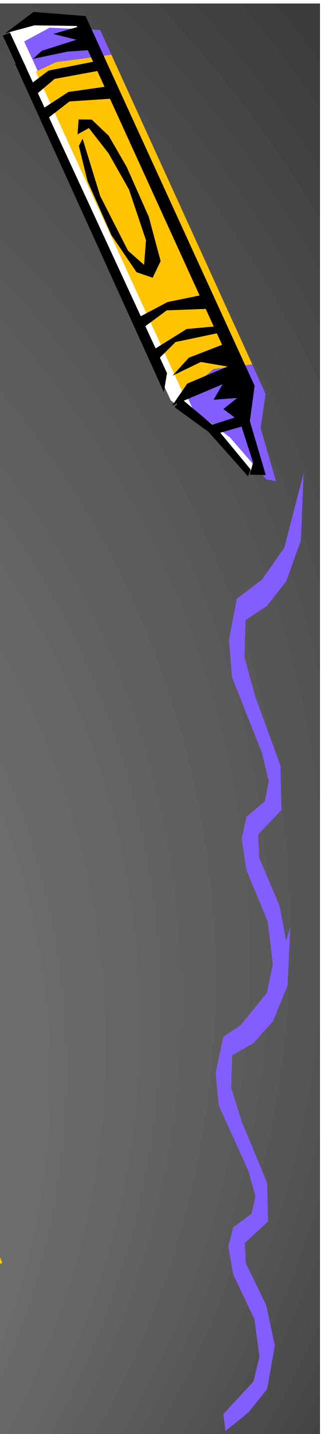
COMPUTING CUBES —INVARIANT INITIALLY HOLD P=>J

P: $\{0 \leqq N\}$

```
k := 0;  r := 0;  s := 1;  t := 6;
    while k≠N do
        a[k] := r;
        r := r + s;
        s := s + t;
        t := t + 6;
        k := k + 1
    end
```

Q: $\{(\forall i \mid 0 \leqq i < N \cdot a[i] = i^3)\}$

J: $\{(\forall i \mid 0 \leqq i < k \cdot a[i] = i^3) \wedge 0 \leqq k \leqq N \wedge r = k^3 \wedge$
$s = 3*k^2 + 3*k + 1 \wedge$
$t = 6*k + 6\}$



COMPUTING CUBES —(1) INVARIANT INITIALLY HOLD P=>J

```
        {$0 \leqq N$} (left)
        {($\forall i \mid 0 \leqq i<k \cdot a[i] = i^3$) $\wedge$ $0 \leqq 0 \leqq N$ ^r = $0^3$ $\wedge$
        $1 = 3*0^2 + 3*0 + 1$ $\wedge$
        $6 = 6*0 + 6$}) (right)
    k := 0;  r := 0;  s := 1;  t := 6;
    J:  {($\forall i \mid 0 \leqq i<k \cdot a[i] = i^3$) $\wedge$ $0 \leqq k \leqq N$ ^r = $k^3$ $\wedge$
        $s = 3*k^2 + 3*k + 1$ $\wedge$
        $t = 6*k + 6$}
     while $k \neq N$ do
              a[k] := r;
              r := r + s;
              s := s + t;
              t := t + 6;
              k := k + 1
      end
        {($\forall i \mid 0 \leqq i<N \cdot a[i] = i^3$)}
    J:  {($\forall i \mid 0 \leqq i<k \cdot a[i] = i^3$) $\wedge$ $0 \leqq k \leqq N$ ^r = $k^3$ $\wedge$
        $s = 3*k^2 + 3*k + 1$ $\wedge$
        $t = 6*k + 6$}
```

COMPUTING CUBES

—(3) J^NOT B=>Q

```
while k≠N do
        a[k] := r;
        r := r + s;
        s := s + t;
        t := t + 6;
        k := k + 1
  end
```

Q: $\{(\forall i \mid 0 \leqq i < N \cdot a[i] = i^3)\}$

J: $\{(\forall i \mid 0 \leqq i < k \cdot a[i] = i^3) \wedge 0 \leqq k \leqq N \wedge r = k^3 \wedge$
$s = 3*k^2 + 3*k + 1 \wedge$
$t = 6*k + 6\}$

J AND NOT B:

COMPUTING CUBES
—) J^NOT B=>Q

```
while $k \neq N$ do
        a[k] := r;
        r := r + s;
        s := s + t;
        t := t + 6;
        k := k + 1
    end
```

Q: $\{(\forall i \mid 0 \leqq i < N \cdot a[i] = i^3)\}$

J: $\{(\forall i \mid 0 \leqq i < k \cdot a[i] = i^3) \wedge 0 \leqq k \leqq N \wedge r = k^3 \wedge$
$s = 3*k^2 + 3*k + 1 \wedge$
$t = 6*k + 6\}$

JAND NOT B: $\{(\forall i \mid 0 \leqq i < N \cdot a[i] = i^3) \wedge r = N^3 \wedge$
$s = 3*N^2 + 3*N + 1 \wedge$
$t = 6*N + 6 \wedge k=N\}$

(J AND NOT B) is stronger than Q, therefore J AND NOT B =>Q

COMPUTING CUBES
—(4) J^B => 0 <=vf

```
while k≠N do
        a[k] := r;
        r := r + s;
        s := s + t;
        t := t + 6;
        k := k + 1
    end
```

vf is N-K

J^B is

COMPUTING CUBES —(5) VF DECREAS: {J^B ^vf=VF} S{vf <VF}

```
while k≠N do
        a[k] := r;
        r := r + s;
        s := s + t;
        t := t + 6;
        k := k + 1
    end
```

J: $\{(\forall i \mid 0 \leqq i < k \cdot a[i] = i^3) \wedge 0 \leqq k \leqq N \wedge r = k^3 \wedge$
$s = 3*k^2 + 3*k + 1 \wedge$
$t = 6*k + 6\}$

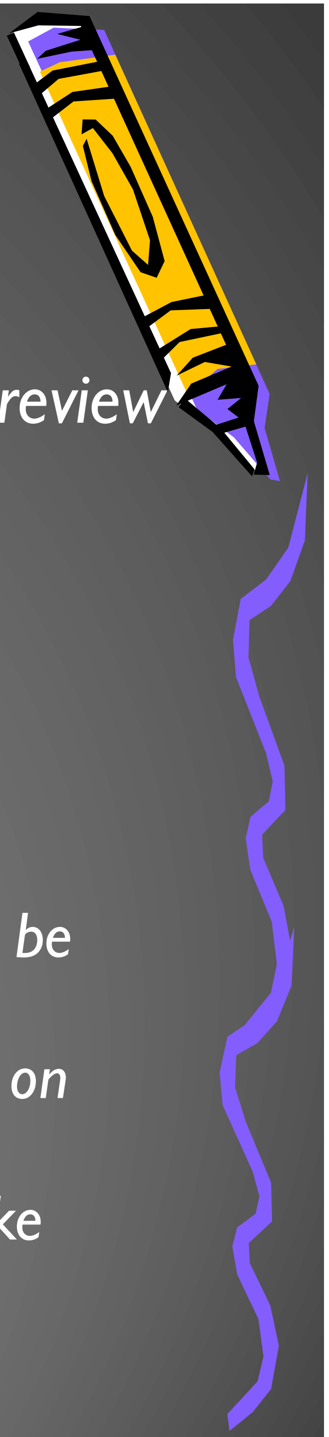
vf is N-K

MODERN CODE REVIEW PRACTICES

MODERN CODE REVIEW PRACTICES

- *Why code review do not find bugs. How current code review practices slows us down.*
- *ICSE 2015, Jacek Czerwonka et al. Microsoft*
  - They minded code review tool usage data
  - CodeFlow is open 5 or 6 hours per developer per day.
  - 30 minutes of interaction time per developer per day.
  - Find defects, improve maintainability, share knowledge, broadcast progress.

MODERN CODE REVIEW PRACTICES



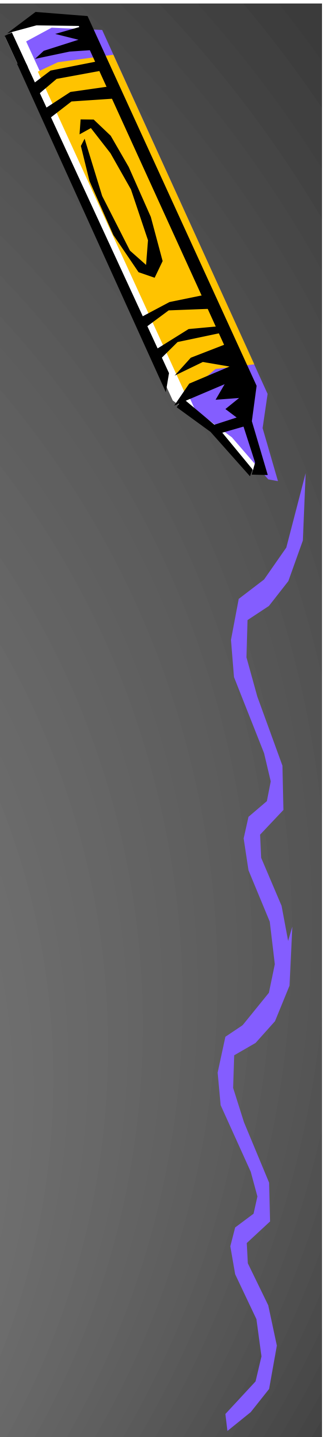
- *Why code review do not find bugs. How current code review practices slows us down.*
- *ICSE 2015, Jacek Czerwonka et al. Microsoft*
  - *Top 3 categories are long term maintenance issues*
    - *comments, naming, and styles (22%)*
    - *Organization of code (16%)*
    - *Alternative solutions for long term maintenance (9%)*
  - *New reviewers learn fast but need at least 6-12 months to be productive as the rest of the team*
  - *People do not actually find bugs – mostly shallow feedback on syntax and style conformance.*
  - *We need to have "rigorous criteria" for code reviews to make them effective.*

TAKE AWAY MESSAGE

- Code reviews are time consuming but developers provide shallow style feedback and are not effective at finding bugs.
- Developers need to have "critical eyes" for reviewing code changes, thinking about corner cases.
- I believe this education exercise of "Hoare Logic" is useful to help my students develop as "effective code reviewers for defect finding."

MORE EXERCISES

COMPUTING CUBES —ANSWERS

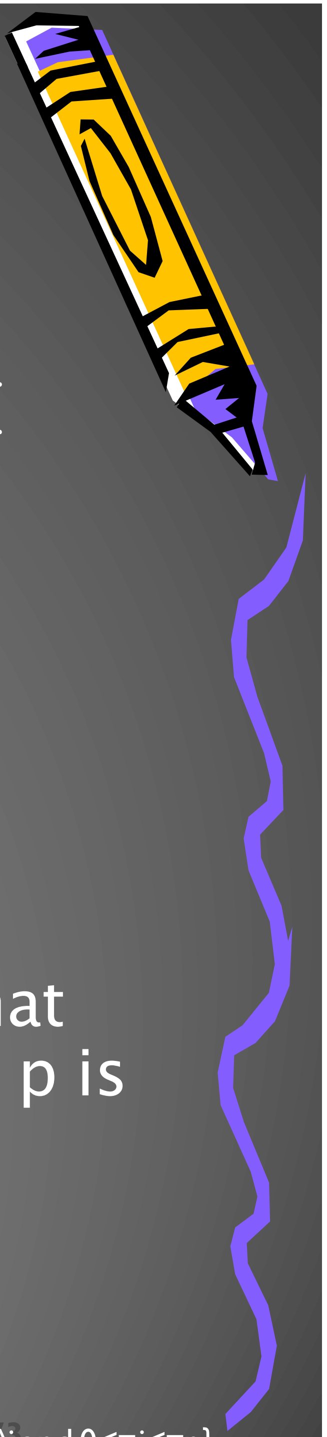


- Invariant:

  $(\forall i \mid 0 \leqq i < k \cdot a[i] = i^3) \land$
  $0 \leqq k \leqq N \land$
  $r = k^3 \land$
  $s = 3*k^2 + 3*k + 1 \land$
  $t = 6*k + 6$
- Variant function:

  N-k
- You proved that this loop terminates!

CLASS EXERCISE: POWER



```
public static int power(int x, int n) {
  int p = 1, i = 0;
  while (i < n) {
      p = p * x;
      i = i + 1;
  }
  return p;
}
```

- Suppose that Sheryl's manager wants a proof that after the execution of the loop, the return value p is equal to x^n.
- What is a loop invariant that must hold given a precondition n>=0?

The loop invariant J is {p=x^i and 0<=i<=n}

- Guess J={ p=x^i AND 0=<i<=n } based on the post condition

(I) P=>J

```
{n>=0}
p = 1, i = 0;
  while (i < n) {
        p = p * x;
        i = i + 1;
  }
```

J={ p=x^i AND 0=<i<=n }

(2) {J AND B} S{ J}

```
p = 1, i = 0;
  while (i < n) {
        p = p * x;
        i = i + 1;
  }
```

(3) J AND NOT =>Q

```
p = 1, i = 0;
  while (i < n) {
        p = p * x;
        i = i + 1;
  }
```

CLASS EXERCISE: QUOTIENT

```
public static int quotient(int n, int d) {
  int q = 0, r = n;
  while (r >= d) {
      r = r - d;
      q = q + 1;
  }
  return q;
}
```

This code performs integer division by repeated subtraction. It divides numerator n by divisor d, returning quotient q and also remainder r. What is a loop invariant that satisfies the expected post-condition, `r<d  AND n = q*d +r given n>=0?`

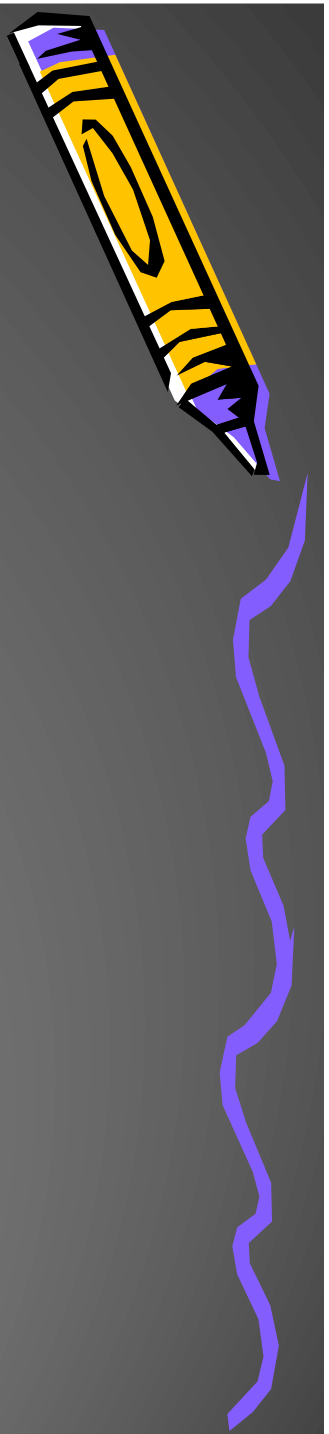
The loop invariant J is {n=qd+r}

- Guess a loop invariant
- $n = qd + r$

(I) P=>J

```
{n>=0}
int q = 0, r = n;
  while (r >= d) {
        r = r - d;
        q = q + 1;
  }
```

J={ n=qd+r}



(2) {J AND B} S{ J}

```
{n>=0}
int q = 0, r = n;
  while (r >= d) {
        r = r - d;
        q = q + 1;
  }
```

J={ n=qd+r}

(3) J AND NOT =>Q

```
{n>=0}
int q = 0, r = n;
  while (r >= d) {
        r = r - d;
        q = q + 1;
  }
```

J={ n=qd+r}

- Derivation to the previous two questions can be found in CCLE. See the scanned notes.

RECAP (1)

- We learned about how to reason about the semantics of assertions, assignment, sequential statements, conditional statements.
- These techniques can help you to find subtle errors during peer code reviews.

RECAP (2)

- Though experienced developers may not use the term, "Hoare Logic or Weakest Preconditions", that's how carefully they analyze the corner cases during peer code reviews.

QUESTIONS?