CS 361, Lecture 6

Jared Saia
University of New Mexico

Today's Outline

- Tons 'o Loop Invariants
- MaxSeq Algorithm
- Sorting?

Loop Invariants

- **Initialization:** Invariant is true before first iteration of loop
- **Maintenance:** If invariant is true before iteration i, it is also true before iteration $i+1$ (for any i)
- **Termination:** When the loop terminates, the invariant gives a property which can be used to show the algorithm is correct

Sum

```
//PRE:
//POST: res is the sum of the elements in arrIn
Sum(int arrIn[], int n)

  int sum = arrIn[0];

  for (int i=1;i<n;i++){
    sum += arrIn[i];
  }
  return sum;
}
```


Sum Invariant

Invariant: At the start of the i-th iteration of the for loop, sum is the summation of arr[0] through arr[i-1]

- **Initialization:** When i=1, sum = arr[0], which establishes the invariant
- **Maintenance:** Assume at the start of the i-th iteration that sum is the summation of arr[0] through arr[i-1]. Then at start of the i+1 iteration, sum is the summation of arr[0] through arr[i-1] plus arr[i]. Thus, sum is the summation of arr[0] through arr[i].
- **Termination:** When the loop terminates, sum is the summation of arr[0] through arr[n-1], which is the sum of the entire array.

Reverse

```
//PRE: n is the size of arrIn
//POST: arrRes is the reverse of arrIn

Reverse(int arrIn[], int n)
  int arrRes[] = new int[n]

  for (int i=0;i<n;i++){
    arrRes[i] = arrIn[(n-1)-i];
  }
  return arrRes;
}
```

Reverse Invariant

Loop Invariant: At the start of the i-th iteration of the for loop, for all $0 \leq j < i$, arrRes[j] = arrIn[(n-1)-j]

- **Initialization:** When i=0, there is no j such that $0 \leq j < i$, so the invariant is trivially true
- **Maintenance:** (Assume at the start of the i-th iteration of the for loop, for all $0 \leq j < i$, arrRes[j] = arrIn[(n-1)-j]. Show that at the start of the i+1 iteration of the for loop, for all $0 \leq j < i+1$, arrRes[j] = arrIn[(n-1)-j]). At the end of the $i+1$ iteration, we know that arrRes[i+1] = arrIn[(n-1)- (i+1)]. This fact, along with the assumption that the invariant holds at the start of the i-th iteration implies that for all $0 \leq j < i+1$, arrRes[j] = arrIn[(n-1)-j])
- **Termination:** When the loop terminates, we know that: for all $0 \leq i < n$, arrRes[i] = arrIn[(n-1)-i]. This implies that arrRes is the reverse of arrIn.

Max

```
//PRE: n is the size of arrIn
//POST: res is the max element in arrIn
Max(int arrIn[], int n)
  int max = arrRes[0]

  for (int i=1;i<n;i++){
    if (arrIn[i]>max)
      max = arrIn[i];
  }
  return max;
}
```


In Class Exercise

Prove that the algorithm Max is correct.

- Give a good loop invariant
- Show the Initialization, Maintenance and Termination conditions for that loop invariant.

MaxSeq

- Proofs of correctness can be very challenging
- Question from before: Design an algorithm to return the largest sum of contiguous integers in an array of ints
- Example: if the input is $(-10, 2, 3, -2, 0, 5, -15)$, the largest sum is 8, which we get from $(2, 3, -2, 0, 5)$.

Our Last Algorithm

```
MaxSeq2 (int arr[], int n)
  int max = 0;
  for (int i = 1;i<=n;i++)
    int sum = 0;
    for (int j=i;j<=n;j++)
        sum += arr[j];
        if (sum > max)
          max = sum;  //and store i and j if desired
  return max;
```

takes $O(n^2)$ time.

A New Algorithm

```
MaxSeq3 (int arr[], int n){

  int arrLeft[] = new int[n];
  arrLeft[0] = arr[0];
  for (int i=1;i<n;i++){
    arrLeft[i] = max (arr[i], arrLeft[i-1] + arr[i]);
  }

  int arrRight[] = new int[n];
  arrRight[n-1] = arr[n-1];
  for (int i=n-2;i>=0;i--){
    arrRight[i] = max (arr[i], arrRight[i+1] + arr[i]);
  }

  ;;now compute the maximum subsequence using
  ;;arrLeft and arrRight
```

```
    int arrMax[] = new int[n];
    arrMax[0] = arrRight[0];
    arrMax[n-1] = arrLeft[n-1];
    for (int i=1;i<n-1;i++){
      int sum = arrLeft[i] + arrRight[i] - arr[i];
      arrMax[i] = sum;
    }
   return the maximum element in the array arrMax or 0,
          whichever is larger;
}
```

Example

arr	-10	2	3	-2	0	5	-15
arrLeft	-10	2	5	3	3	8	-7
arrRight	-2	8	6	3	5	5	-15
arrMax	-2	8	8	8	8	8	-7

MaxSeq3

- What is the run time of this algorithm?
- Is it correct?

Loop1 Invariant

- *Loop 1 Invariant: At the start of the i-th iteration, for all $0 \leq j < i$, arrLeft[j] gives the largest value of any subsequence whose rightmost term is $arr[j]$.*
- **Initialization:** When $i = 1$, arrLeft[0] = arr[0], which is the largest value of any subsequence whose rightmost term is arr[0].
- **Maintenance:** Assume the invariant is true before iteration i. This means arrLeft[i-1] gives the value of the largest subsequence whose rightmost term is arrLeft[i-1].
  Note that at the end of the iteration, arrLeft[i] = max (arr[i], arrLeft[i-1] + arr[i]). Further note that there exists a subsequence, l_i* which terminates at arr[i] and obtains this value. It's either the subsequence consisting of just arr[i], or the subsequence with term arr[i] concatenated with the subsequence associated with the value arrLeft[i-1].
  Now consider some arbitrary subsequence, l_i which has rightmost term arr[i]. Let $v(l_i)$ be the value of this subsequence.

  To show arrLeft[i] is indeed the maximal value, we need only show that that $v(l_i) \leq$ arrLeft[i]. There are two cases.
  Case 1 is that l_i includes only the term arr[i]. In this case, $v(l_i) \leq$ arr[i] $\leq$ arrLeft[i].
  Case 2 is that l_i extends left beyond arr[i]. Let l_{i-1} be the part of l_i that does not contain arr[i]. Then $v(l_i) = v(l_{i-1}) +$ arr[i]. But $v(l_{i-1}) \leq$ arrLeft[i-1], by the inductive hypothesis. Thus $v(l_i) \leq v$ arrLeft[i-1] + arr[i] $\leq$ arrLeft[i].
  Hence the value arrLeft[i] does in fact give the largest value of any subsequence whose rightmost term is arr[i], so by the inductive hypothesis, the loop invariant holds after iteration i.
- **Termination:** When the loop terminates, for all values of $0 \leq j < n$, arrLeft[j] gives the largest value of any subsequence whose rightmost term is arr[j].

Loop2 Invariant

- *Loop 2 Invariant: At the start of the i-th iteration, arrRight[j] gives the value of the largest subsequence whose leftmost term is $arr[j]$, for all $n > j > i$.*
- **Initialization**, **Maintenance**, and **Termination** proofs are similar to Loop 1 invariant
- Good at home exercise to see if you can prove these facts for loop2

Loop3 Invariant

- *Loop 3 Invariant: At the start of the i-th iteration, for all $j < i$, arrMax[j] gives the value of the best subsequence which includes value arr[j].*
- We can assume the termination conditions of the loop1 and loop2 invariants hold during loop3.
- **Initialization:** When $i = 1$, arrMax[0] = arrRight[0]. We've shown that arrRight[0] is the best value of any subsequence whose leftmost value is arr[0]. Any subsequence containing arr[0] will have arr[0] as the leftmost element. Hence arrMax[0] is in fact the value of the best subsequence containing arr[0].
- **Maintenance:** Assume the invariant is true before iteration i. Note that at the end of the iteration, arrMax[i] = arrLeft[i] + arrRight[i] - arr[i].
  We first note that there exists a subsequence s_i* which achieves this value arrMax[i]. It's just the subsequence consisting

of the subsequence which achieves the value arrLeft[i] concatenated with the subsequence which achieves the value arrRight[i].
Now consider some arbitrary subsequence, s_i, which contains arr[i]. To show arrMax[i] is indeed the maximal value, we need only show that $v(s_i) \leq$ arrMax[i]. Let l_i be the subsequence of s_i which includes arr[i] and all elems to the left of arr[i]. Similarly, let r_i be the subsequence of s_i which includes arr[i] and all elems to the *right* of arr[i]. Note that
– $v(l_i) \leq$ arrLeft[i]
– $v(r_i) \leq$ arrRight[i]
Hence $v(s_i) = v(l_i) + v(r_i) -$ arr[i] $\leq$ arrLeft[i] + arrRight[i] -arr[i] = arrMax[i]. And so arrMax[i] does in fact give the value of the best subsequence which includes value arr[i]. Thus, the loop invariant remains true at the beginning of iteration $i + 1$.

- **Termination:** When the loop terminates, for all $1 < j < n-1$, arrMax[j] gives the value of the best subsequence which includes value arr[j]. We further note that arrMax[n-1] gives the value of the best subsequence containing arr[n-1], since arrMax[n-1] = arrLeft[n-1], and any subsequence containing arr[n-1] will have arr[n-1] as the rightmost element.
  The best subsequence in the array arr must contain some element in the array or be the empty subsequence. If it's not the empty subsequence, the value of it is stored somewhere in arrMax. Thus the return value of MaxSeq3 is the value of the best possible subsequence.

Take away

- We needed 3 loop invariants for MaxSeq3
- MaxSeq3 was much harder to show correct, but it runs *much* faster than our other algorithms
- I don't expect you to be able to do the entire proof for MaxSeq3, especially not from scratch
- However, you should be able to understand and do something similar to the individual loop invariant proofs
- Also, you should be able to understand the entire proof!

The Sorting Problem

- The Problem: we want to sort an array, A, of integers in non-decreasing order
- E.g. if A is $3, 2, 2, 1, 5$ at the start, we want it to be $1, 2, 2, 3, 5$ at the end
- Sorting is a *very* common programming problem!
- Last time, we analyzed the Insertion-Sort Algorithm

Insertion Sort

Insertion-Sort (A, int n)

```
for (j=1;j<n;j++){
  key = A[j];
  //Insert A[j] into the sorted sequence A[0,...,j-1],
  //in the location such that it is as large as all elems
  // to the left of it
  i = j-1
  while (i>=0 and A[i] > key){
    A[i+1] = A[i]
    i--
  }
  A[i+1] = key
}
```

Analysis

- Best case run time of Insertion Sort is $O(n)$ (if the array is already sorted)
- However, we proved last time that the run time of Insertion Sort is $\Theta(n^2)$ in the worst case
- Q: Can we do better than this?
- A: Yes, we can use a recursive algorithm called Merge Sort

Merge Sort

High Level Idea:

- Split the array into two parts of the same size, A_1 and A_2
- Recursively sort A_1 and A_2
- Merge A_1 and A_2 together into one big sorted array

Merge Sort

```
//POST: res[]
Merge-Sort (int A[])
  int arrRes[] = A;
  if (A.size() > 1){
    //set m to be the ``middle'' of the array
    m = floor (A.size()/2);
    int arrLeft[] = A[0,..,m]
    int arrRight[] = A[m+1,..,A.size()]
    arrLeft = Merge-Sort (arrLeft);
    arrRight = Merge-Sort (arrRight);
    arrRes = Merge (arrLeft,arrRight);
  }
   return arrRes;
}
```


Merge

```
//PRE: arrLeft and arrRight are in sorted order
//POST: arrRes contains the elems of arrLeft and arrRight
//      in sorted order
Merge(int arrLeft[], int arrRight[])
  iLeft = iRight = 0;
  int arrRes[] = new int[arrLeft.size()+ arrRight.size()];
  for (int i=0;i<arrRes.size();i++){
    if (iRight == arrRight.size () ||
         (iLeft<arrLeft.size()
          && arrLeft[iLeft]<=arrRight[iRight])){
      arrRes[i] = arrLeft[iLeft];
      iLeft++;
    }else{
      arrRes[i] = arrRight[iRight];
      iRight++;}
  }
  return arrRes;
}
```


Merge Example

arrLeft	1	4	5			
arrRight	2	3	6			
arrRes	1	2	3	4	5	6

Todo

- Read Chapter 4 (Recurrences) in text