# CS 361, Lecture 5

Jared Saia
University of New Mexico

## Today's Outline

- Review of Asymptotic Notation
- Proofs of correctness
- Intro to Recurrence Relations



## Asymptotic Rule of Thumb

- Let $f(n)$, $g(n)$ be two functions of $n$
- Let $f_1(n)$, be the fastest growing term of $f(n)$, stripped of its coefficient.
- Let $g_1(n)$, be the fastest growing term of $g(n)$, stripped of its coefficient.

Then we can say:

- If $f_1(n) \geq g_1(n)$ then $f(n) = O(g(n))$
- If $f_1(n) \leq g_1(n)$ then $f(n) = \Omega(g(n))$
- If $f_1(n) = g_1(n)$ then $f(n) = \Theta(g(n))$
- If $f_1(n) < g_1(n)$ then $f(n) = o(g(n))$
- If $f_1(n) > g_1(n)$ then $f(n) = \omega(g(n))$



## Examples

The following are all true statements:

- From last lecture, $\sum_{i=1}^{n} i^2$ is $O(n^3)$, $\Omega(n^3)$ and $\Theta(n^3)$
- $\log n$ is $o(\sqrt{n})$
- $\log n$ is $o(\log^2 n)$
- $10,000n^2 + 25n$ is $\Theta(n^2)$



## Proofs of Correctness

- Last time, we saw how to use a loop invariant to prove the correctness of an alg to find the middle element in a list
- Today we'll look at proof of correctness for Insertion Sort
- We'll also look at a more complicated proof for a MaxSeq algorithm



## Loop Invariants

A useful tool for proofs of correctness is loop invariants. Three things must be shown about a loop invariant

- **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

## Example Loop Invariant

- We'll prove the correctness of a simple algorithm which solves the following interview question:
- *Find the middle of a linked list, while only going through the list once*
- The basic idea is to keep two pointers into the list, one of the pointers moves twice as fast as the other
- (Call the head of the list the 0-th elem, and the tail of the list the $(n-1)$-st element, assume that $n-1$ is an even number)



## Example Algorithm

```
  GetMiddle (List l){
    pSlow = pFast = l;
    while ((pFast->next)&&(pFast->next->next)){
      pFast = pFast->next->next
      pSlow = pSlow->next
    }
    return pSlow
}
```



## Example Loop Invariant

- *Invariant: At the start of the $i$-th iteration of the while loop, pSlow points to the $i$-th element in the list and pFast points to the $2i$-th element*
- **Initialization:** True when $i = 0$ since both pointers are at the head
- **Maintenance:** if pSlow, pFast are at positions $i$ and $2i$ respectively before $i$-th iteration, they will be at positions $i+1$, $2(i+1)$ respectively before the $i+1$-st iteration
- **Termination:** When the loop terminates, pFast is at element $n-1$. Then by the loop invariant, pSlow is at element $(n-1)/2$. Thus pSlow points to the middle of the list



## Another example

- 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
- Insertion-sort is one way to do this



## 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
}
```



## Run Time of Insertion-Sort

- We can easily calculate the worst case run time of insertion sort
- The outer "for" loop runs from $j = 1$ to $j = n-1$, in the worst case, the inner "while" loop runs from $i = j-1$ to 0
- This gives us the following sum:

$$\sum_{j=1}^{n} \sum_{i=j-1}^{0} 1 = \sum_{j=1}^{n} j \tag{1}$$

$$= (n+1)n/2 \tag{2}$$

$$= O(n^2) \tag{3}$$

$$\tag{4}$$

## Loop Invariant

- Insertion sort has a more complicated loop invariant:
- *Invariant: At the start of each iteration of the for loop, the array* $A[0, \ldots, j-1]$ *consists of the elements of the original* $A[0, \ldots, j-1]$*, except that they are in sorted order*
- How do we use this loop invariant to prove correctness?



## In Class Exercise

- *Invariant: At the start of each iteration of the for loop, the array* $A[0, \ldots, j-1]$ *consists of the elements of the original* $A[0, \ldots, j-1]$*, except that they are in sorted order*
- Establish the following properties for this invariant:
  - Initialization: Establish at time just after first assignment to $j$ (i.e. for $j = 1$, but before the loop has been entered)
  - Maintenance: Assuming the inner loop does what the comment says, show maintenance for the outer loop invariant
  - Termination: Show that $A$ is sorted at termination



## Another Example

- Proofs of correctness are not always easy
- 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 $j < i$, 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 ending at position $j$, for all $j < i$.*
- **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 proofs like the one for MaxSeq3, especially not from scratch
- However, you should be able to understand it!
- I *do* expect you to be able to do proofs of correctness like those for GetMiddle and Insertion-Sort.



## Todo

- Read Chapter 4 (Recurrences) in text