

Analysis of Algorithms CS 3343 Lecture Three

Prof. William Winsborough
September 9, 2008

Business

- Recall: Homework 1 due Thursday September 11
 - Exercises 2.1-3, 2.2-1, 2.2-2, 2.3-1, 2.3-3, 2.3-5
 - Needless to say, I meant to ask you to read sections 2.2, 2.3
- Look over section 3.2
- Read section 4.1
- In the future, we will reverse the order of lecture and recitation on Tuesdays
 - The 3:30 – 4:45 period will be used for a combination of recitation and lecture
 - The 5:30 – 6:20 period will be used for lecture

Lecture outline

- Review some of the material presented by Prof. Ruan
- Investigate more deeply how to prove loop invariants

Using loop invariants to prove correctness of loops

- A loop invariant is a formal statement
 - It relates the values of variables at specific points during execution of the algorithm
 - It must hold
 - Before the loop is entered
 - At the end of each iteration
- Technically, a loop invariant is an induction hypothesis

Using loop invariants to prove correctness of loops

- We prove that the loop invariant holds by using induction on the number of iterations performed so far
 - Initialization: the invariant is true prior to the 1st iteration
 - Maintenance: if the invariant is true before the j^{th} iteration, it remains true before the $(j+1)^{\text{th}}$ iteration
 - Termination: when the loop terminates, it does so due to some *exit condition*
 - This, together with the invariant, states what the loop has achieved
 - This is the goal for which the invariant is designed

Prove correctness using loop invariants

```
InsertionSort(A, n) {  
  for j = 2 to n {  
    key = A[j];  
    i = j - 1;  
    ▶Insert A[j] into the sorted sequence A[1..j-1]  
    while (i > 0) and (A[i] > key) {  
      A[i+1] = A[i];  
      i = i - 1;  
    }  
    A[i+1] = key;  
  }  
}
```

Loop invariant: at the start of each iteration of the for loop, the subarray consists of the elements originally in A[1..j-1] but in sorted order.

Initialization

```

InsertionSort(A, n) {
  for j = 2 to n {
    key = A[j];
    i = j - 1;
    >Insert A[j] into the sorted sequence A[1..j-1]
    while (i > 0) and (A[i] > key) {
      A[i+1] = A[i];
      i = i - 1;
    }
    A[i+1] = key;
  }
}

```

Subarray A[1] is sorted. So loop invariant is true before the loop starts.

9 September 2008 Winsborough CS 3343 Lecture 3 7

Maintenance

```

InsertionSort(A, n) {
  for j = 2 to n {
    key = A[j];
    i = j - 1;
    >Insert A[j] into the sorted sequence A[1..j-1]
    while (i > 0) and (A[i] > key) {
      A[i+1] = A[i];
      i = i - 1;
    }
    A[i+1] = key;
  }
}

```

Assume loop variant is true prior to iteration j

Loop variant will be true before iteration j+1

9 September 2008 Winsborough CS 3343 Lecture 3 8

How to Prove the Invariant is Maintained?

- Need to show that the while loop does the right thing
 - Another loop invariant:
 - A[1..i] is sorted
 - A[i+2..j] is sorted (initially this subarray is empty)
 - $i+2 \leq j \rightarrow \text{key} < A[i+2]$
 - $i+2 \leq j \rightarrow A[i] < A[i+2]$
 - Loop exit condition: $i \leq 0$ or $A[i] \leq \text{key}$
- When $i \leq 0$, we actually have $i = 0$
 - Proving this is somewhat tricky
 - The invariant tells us $A[i+2..j]$ is sorted and $\text{key} < A[i+2]$, so assigning $A[i+1] = \text{key}$ gives us that $A[1..j]$ is sorted
- When $A[i] \leq \text{key}$
 - We use $\text{key} < A[i+2]$, together with $A[i] < A[i+2]$, $A[1..i]$ is sorted, and $A[i+2..j]$ is sorted to show that $A[1..j]$ is sorted

9 September 2008 Winsborough CS 3343 Lecture 3 9

Termination

```

InsertionSort(A, n) {
  for j = 2 to n {
    key = A[j];
    i = j - 1;
    >Insert A[j] into the sorted sequence A[1..j-1]
    while (i > 0) and (A[i] > key) {
      A[i+1] = A[i];
      i = i - 1;
    }
    A[i+1] = key;
  }
}

```

Upon termination, A[1..n] contains the original elements in sorted order.

9 September 2008 Winsborough CS 3343 Lecture 3 10

Comparison of functions

n	log ₂ n	n	nlog ₂ n	n ²	n ³	2 ⁿ	n!
10	3.3	10	33	10 ²	10 ³	10 ³	10 ⁶
10 ²	6.6	10 ²	660	10 ⁴	10 ⁶	10 ³⁰	10 ¹⁵⁸
10 ³	10	10 ³	10 ⁴	10 ⁶	10 ⁹		
10 ⁴	13	10 ⁴	10 ⁵	10 ⁸	10 ¹²		
10 ⁵	17	10 ⁵	10 ⁶	10 ¹⁰	10 ¹⁵		
10 ⁶	20	10 ⁶	10 ⁷	10 ¹²	10 ¹⁸		

For a super computer that does 1 trillion operations per second. It will longer than 1 billion years

9 September 2008 Winsborough CS 3343 Lecture 3 12

Asymptotic notations

- O: Big-O
- Ω: Omega
- Θ: Theta

9 September 2008 Winsborough CS 3343 Lecture 3 12

Formal definitions

- $f(n) \in O(g(n))$
 - There exist positive constants c and n_0 such that for all n , $n_0 < n \rightarrow 0 \leq f(n) \leq cg(n)$
- $f(n) \in \Omega(g(n))$
 - There exist positive constants c and n_0 such that for all n , $n_0 < n \rightarrow 0 \leq cg(n) \leq f(n)$
- $f(n) \in \Theta(g(n))$
 - $f(n) = O(g(n))$ and $f(n) = \Omega(g(n))$
 - Equivalently: There exist positive constants c_1 , c_2 , and n_0 such that for all n , $n > n_0 \rightarrow 0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n)$

9 September 2008

Winsborough CS 3343 Lecture 3

13

Divide and Conquer

- General idea:
 - Divide a large problem into smaller ones
 - By a constant ratio
 - By a constant or some variable
 - Solve each smaller one *recursively* or *explicitly*
 - Combine the solutions of smaller ones to form a solution for the original problem

9 September 2008

Winsborough CS 3343 Lecture 3

14

Merge sort

MERGE-SORT $A[1..n]$

1. If $n = 1$, done.
2. Recursively sort $A[1.. \lceil n/2 \rceil]$ and $A[\lceil n/2 \rceil + 1..n]$.
3. “Merge” the 2 sorted lists.

Key subroutine: MERGE

9 September 2008

Winsborough CS 3343 Lecture 3

15

How to show the correctness of a recursive algorithm?

- By induction:
 - Base case: prove it works correctly when the input does not require the use of recursion
 - Inductive hypothesis: assume the solution is correct for all recursively solved sub-problems
 - Step: show that, if the induction hypothesis is correct, then the algorithm is correct for the original problem

9 September 2008

Winsborough CS 3343 Lecture 3

16

Correctness of merge sort

MERGE-SORT $A[1..n]$

1. If $n = 1$, done.
2. Recursively sort $A[1.. \lceil n/2 \rceil]$ and $A[\lceil n/2 \rceil + 1..n]$.
3. “Merge” the 2 sorted lists.

Proof:

1. Base case: if $n = 1$, the algorithm will return the correct answer because $A[1..1]$ is already sorted.
2. Inductive hypothesis: assume that the algorithm correctly sorts $A[1.. \lceil n/2 \rceil]$ and $A[\lceil n/2 \rceil + 1..n]$.
3. Step: if $A[1.. \lceil n/2 \rceil]$ and $A[\lceil n/2 \rceil + 1..n]$ are both correctly sorted, the whole array $A[1.. \lceil n/2 \rceil]$ and $A[\lceil n/2 \rceil + 1..n]$ is sorted after merging.

9 September 2008

Winsborough CS 3343 Lecture 3

17

How to Prove the Step?

- In the case of Merge, this will be another induction
- Loop invariant says
 - The values merged so far are in sorted order
 - They are all less than or equal to the values remaining in the two recursively sorted list

9 September 2008

Winsborough CS 3343 Lecture 3

18

How to analyze the time-efficiency of a recursive algorithm?

- Express the running time on input of size n as a function of the running time on smaller problems
 - Such equalities are called *recurrences*

Analyzing merge sort

$T(n)$ | **MERGE-SORT** $A[1 \dots n]$
 $\Theta(1)$ | 1. If $n = 1$, done.
 $2T(n/2)$ | 2. Recursively sort $A[1 \dots \lceil n/2 \rceil]$
 | and $A[\lceil n/2 \rceil + 1 \dots n]$.
 $C(n)$ | 3. "Merge" the 2 sorted lists

Sloppiness: Should be $T(\lceil n/2 \rceil) + T(\lfloor n/2 \rfloor)$, but it turns out not to matter asymptotically.

Analyzing merge sort

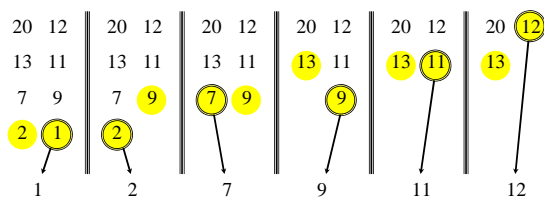
- Divide:* Trivial.
- Conquer:* Recursively sort 2 subarrays.
- Combine:* Merge two sorted subarrays

$$T(n) = 2T(n/2) + D(n) + C(n)$$

subproblems subproblem size work dividing work combining

- What is the time for the base case? **Constant**
- What are $D(n)$ and $C(n)$?
- What is the growth order of $T(n)$?

Merging two sorted arrays



$\Theta(n)$ time to merge a total of n elements (linear time).

Recurrence for merge sort

$$T(n) = \begin{cases} \Theta(1) & \text{if } n = 1; \\ 2T(n/2) + \Theta(n) & \text{if } n > 1. \end{cases}$$

- Note that $\Theta(n)$ stands for an arbitrary unknown function belonging to $\Theta(n)$
- The base case is often not mentioned when $T(n) = \Theta(1)$ for values of n that are bounded by a constant

- What function $T(n)$ satisfies this equation?
- And what is its asymptotic complexity?
- Do problem 2.3-3 and stay tuned on Thursday