Algorithms and Data Structures

Chapter 2

Catherine Durso

`cdurso@cs.du.edu`

Chapter 2

Chapter 2 provides an overview of the analysis of an algorithm correctness and the analysis of an algorithms's asymptotic running time. The INSERTION SORT and MERGE SORT algorithms for sorting arrays are used as examples.
These algorithms are presented using pseudocode. The text's version of pseudocode uses indentation in place of nested brackets and uses "//" to introduce comments. Other conventions are detailed on pages 20 and 21.

Insertion Sort

```
Insertion-Sort(A)
  for j=2 to A.length
    key=A[j]
    // Insert A[j] into the sorted sequence
    i=j-1
    while i>0 and A[i]>key
      A[i+1]=A[i]
      i=i-1
    A[i+1]=key
```

Correctness

When you specify an algorithm in pseudocode but do not implement it, how can you verify that the algorithm is correct? Correctness becomes more difficult to verify when the algorithm involves iteration.

In this case, one approach is to use loop invariants. A loop invariant is a statement about the state of the program, including the data, each time control passes to the loop statement.

Loop Invariant

- The loop invariant should be true when control first passes to the loop. (*Initialization*)
- If the loop invariant was true before an iteration of the loop block, and the loop condition is true, then the loop invariant is true after the iteration, at the evaluation of the loop condition. (*Maintenance*)
- The loop invariant should give useful information about the configuration of the program when control exits the loop. (*Termination*)

For- loop Invariant

For now, assume that if the while-loop is passed an array A with $A[1,..j-1]$ in ascending order, then after line 8, the array A is a permutation of the the input array, and $A[1,..j]$ is in ascending order.

The loop invariant for the for-loop is "A is a permutation of the input array, and $A[1,..j-1]$ is in ascending order."

Invariant Verification

Initialization j=2: The input array has not been changed. The claim that $A[1,..j-1]$ in ascending order becomes the claim that $A[1,..2-1] = A[1]$ is in ascending order is trivially true.

cont.

Maintenance: We are assuming that the effect of the while-loop with line 8, when passed an array with $A[1,..j-1]$ in ascending order, is to return a permutation of that array with $A[1,..j]$ in ascending order. Thus if the loop invariant and the loop condition are true at line 1, then at line 8, A is a permutation of the input array, and $A[1,..j]$ is in ascending order. Thus the loop invariant remains true after the incrementation of j in the for-loop.

cont.

Termination: The for-loop terminates with the loop invariant true and $j = A.length + 1$. Thus $A[1, ..A.length]$ is in ascending order and is a permutationof the input array, as required.

While-loop

Each iteration of the while-loop compares the key, the value in $A[j]$, to the value in $A[i]$, and copies value in $A[i]$ one position up if $A[i] > key$. The goal is to verify the assumption used in analysis of the for-loop. Thus the following is a reasonable loop invariant: A, with key inserted into $A[i+1]$ is a permutation of the input array for the loop. The values in $A[1,..i]$ followed by the values in $A[i+2,..j]$ are the values in the input array in positions $1,..j-1$, in the input order. Further, $A[k] > key$ for $k \in \{i+2,..j\}$.

Initialization

At this point, $i = j - 1$, and the input array has not been changed. Thus A with key inserted into $A[i+1] = A[j]$ is a permutation of the input array for the loop. The subarray $A[i+2, ..j]$ is empty, so the values in $A[1, ..i]$ followed by the values in $A[i+2, ..j]$ are the values in the input array in positions $1, ..j-1$, in the input order. The set $\{i+2, ..j\}$ is empty, so the comparison $A[k] > key$ for $k \in \{i+2, ..j\}$ is trivially true.

Maintenance

Given that the loop condition is true and the loop is entered, $A[i] > key$. Line 6 copies $A[i]$ into $A[i+1]$. Thus the values in $A[1, ..i-1]$ followed by the values in $A[i+1, ..j]$ are the values in the input array in positions $1, ..j-1$, in the input order. Also, $A[k] > key$ for $k \in \{i+1, ..j\}$. In line 7, i is decremented. For this new i, the values in $A[1, ..i]$ followed by the values in $A[i+2, ..j]$ are the values in the input array in positions $1, ..j-1$, in the input order. Also, $A[k] > key$ for $k \in \{i+2, ..j\}$, as required.

Termination

The loop terminates with the loop condition true and $A[i] \leq key$. Line 8 copies key into $A[i+1]$. The loop invariant implies that $A[1,..j]$ contains the values in in the input array in positions $1,..j$, A is a permutation of the input array, and the values in $A[i+2,..j]$ are greater that key, the value in position $i+1$. Assuming that $A[1,..j-1]$ was in ascending order when the while-loop was entered, and A contains a permutation of the values in the input array for the function, $A[1,..j]$ is in ascending order after line 8 and A is still a permutation of the argument of the function.

Analyzing Algorithms

Algorithm analysis has come to mean prediction of the resources that an algorithm requires. Often the resource in question is time. Other considerations include disk accesses and memory. Emphasis in this course is on time.

In order to study the time required by an algorithm we need a model for the resource demands of the steps in the algorithm. We will generally assume a single processor with all memory operations taking place in RAM.

Basic operations include add, subtract, multiply, divide, call and return from subroutines, branch, and load, store, and copy single values.

T(n)

We will focus on running time as a function of input size, $T(n)$ where n is the size of the input. We are primarily concerned with the growth of $T(n)$ as n becomes very large.

- The concept of size is problem-specific.
- The running time may be different for different instances of the same size. This leads to consideration of worst-case and average case running times.
- Behavior of $T(n)$ for large n is basically described in terms of ratios, such as $max\left\{\frac{T(n)}{n^2}|n \in \mathbb{Z}\right\} < \infty$ or $lim_{n\to\infty}\frac{T(n)}{n^2} = l$. We will formalize this in Chapter 3.

Analysis Example

We will calculate $T(n)$ for Insertion-Sort, using constants $c_1, ...c_8$ for the time required to run each line a single time. The number of times the while-condition is tested depends on j and on the instance. Denote this by t_j.

times

Insertion-Sort(A)

		cost	times
1	for j=2 to A.length	c_1	n
2	key=A[j]	c_2	$n-1$
3	// Insert A[j] into the sorted A[1..j-1]	0	
4	i=j-1	c_4	$n-1$
5	while i>0 and A[i]>key	c_5	$\sum_{j=2}^{n} t_j$
6	A[i+1]=A[i]	c_6	$\sum_{j=2}^{n} (t_j - 1)$
7	i=i-1	c_7	$\sum_{j=2}^{n} (t_j - 1)$
8	A[i+1]=key	c_8	$n-1$

total time

$T(n) = c_1 n + c_2\,(n-1) + c_4\,(n-1) + c_5 \sum_{j=2}^{n} t_j + c_6 \sum_{j=2}^{n}(t_j - 1) + c_7 \sum_{j=2}^{n}(t_j - 1) + c_8\,(n-1)$

Note $1 \le t_j \le j$.

Under what circumstances is $t_j = 1$? $t_j = j$?

Worst case

$T(n) = c_1 n + c_2 (n-1) + c_4 (n-1) + c_5 \sum_{j=2}^{n} j + c_6 \sum_{j=2}^{n} (j-1) + c_7 \sum_{j=2}^{n} (j-1) + c_8 (n-1)$

Note $\sum_{j=2}^{n} j = \frac{n(n+1)}{2} - 1$ and $\sum_{j=2}^{n} (j-1) = \frac{n(n-1)}{2}$.

Using this and simplifying, the worst case for $T(n)$ is

$T(n) = \frac{c_5+c_6+c_7}{2} n^2 + \left(c_1 + c_2 + c_4 - \frac{-c_5+c_6+c_7}{2} + c_8\right) n - (c_2 + c_4 + c_5 + c_8)$

Order of Growth

Because the function $T(n)$ for the worst-case running time of INSERTION-SORT satisfies $lim_{n\to\infty}\frac{T(n)}{n^2} = l > 0$, we say that INSERTION-SORT has a worst-case running time that is $\Theta\left(n^2\right)$. (More on this in Chapter 3.)

Average Case

In order to be able to calculate the average case running time or *expected running time* for an algorithm, we need to know the distribution of instances to which the algorithm will be applied. We may make an assumption about this, or use randomization to produce a known distribution of input instances.

For example, in the case of INSERTION-SORT, we could assume that all values in A are distinct and all permutations of their ranks are equally likely.

big-O

The worst case running time is a bound on the average case. Thus for INSERTION-SORT we know that if $T(n)$ is the expected running time for some distribution of inputs, then $max\left\{\frac{T(n)}{n^2}|n \in \mathbb{Z}\right\} < \infty$. This allows us to say that the expected running time for INSERTION-SORT is $O(n^2)$.

Order of Growth

Because the Θ- notation is not sensitive to the size of the constants, a program implementing a $\Theta(n^2)$ algorithm may be faster that, say, a program implementing a $\Theta(n \log(n))$ algorithm for small inputs. But, for large enough inputs, the $\Theta(n \log(n))$ will be faster.

Design Methods

Incremental Design, or Decrease and Conquer: Solve a problem of size n by processing the n components of the input sequentially. The INSERTION-SORT algorithm has an incremental design. We position $A[2]$, then $A[3]$, etc.

Recursive Design, or Divide and Conquer: Divide a problem of size n into smaller problems of the same type. Conquer the smaller problems by solving recursively, or, for small instances, solving directly. Combine the solutions of the smaller problems to obtain a solution of the original instance. MERGE-SORT is an example of this.

Merge Sort

The essential observation for MERGE-SORT is that if you have two sorted arrays, you can quickly merge the values into a third, sorted array. This enables us to create a divide-and-conquer sort, in which we divide the input array into two subarrays, sort these recursively, and merge the results.

The MERGE routine mimics the action of merging two sorted piles of cards by repeatedly taking the smallest card showing off its stack and placing it face-down in the output pile. When both sorted piles are empty, the output pile is a sorted pile of all the cards.

Merge basics

The $\textsc{Merge}$ in the text takes an array A with $p \leq q \leq r$ indices in A. Assuming the subarrays $A[p..q]$ and $A[q+1..r]$ are sorted, the routine sorts $A[p..r]$. The routine copies $A[p..q]$ and $A[q+1..r]$ into auxilliary arrays, using ∞ as a sentinel value at the end of each of the auxilliary arrays.

Merge routine

```
MERGE($A$, $p$, $q$, $r$)
1    $n_1 = q - p + 1$
2    $n_2 = r - q$
3    Let $L[1..n_1 + 1]$ and $R[1..n_2 + 1]$ be new arrays.
4    for $i = 1$ to $n_1$//Copy $A\,[p..q]$ into $L$
5            $L[i] = A[p + i - 1]$
6    for $j = 1$ to $n_2$//Copy $A\,[q + 1..r]$ into $R$
7            $R[i] = A[q + j]$
8    $L[n_1 + 1] = \infty$
9    $R[n_2 + 1] = \infty$
10   $i = 1$
11   $j = 1$
12   for $k = p$ to $r$
13           if $L[i] \leq R[j]$
14                   $A[k] = L[i]$
15                   $i = i + 1$
16           else $A[k] = R[j]$
17                   j=j+1
```

Merge Sort

```
MERGE-SORT($A$, $p$, $r$)
1 if $p < r$
2         $q = \lfloor (p + r) / 2 \rfloor$ //Remember floors? ceilings?
3          MERGE-SORT ($A$, $p$, $q$)
4          MERGE-SORT ($A$, $q + 1$, $r$)
5          MERGE($A$, $p$, $q$, $r$)
```

(This could have been written iteratively, but recursion yields shorter pseudocode.)

Merge loop invariant

The goal is to show that, assuming the subarrays $A[p..q]$ and $A[q+1..r]$ are sorted, the routine sorts $A[p..r]$.

At the loop-test in the for-loop at line 12, $A[p..k-1]$ contains the $k-p$ smallest elements of $A[p..r]$in sorted order. $L[i]$and $R[j]$are the smallest elements in the respective arrays that are not in $A[p..k-1]$.

Initialization

When the loop is entered, $k = p$ and $i = j = 1$. $A[p..k-1] = A[p..p-1] = \emptyset$, so it trivially contains the $k - p = p - p = 0$ smallest elements of $A[p..r]$in sorted order. On valid input, $L[1]$ and $R[1]$ are the smallest elements in their respective arrays, and so are the smallest elements in the respective arrays that are not in $A[p..p-1]$.

Maintenance

Consider the case $R[j] < L[i]$. The 'else' will be executed, copying $R[j]$ into A in position k and incrementing j. Now $A[p..k]$ contains the $k-p+1$ smallest elements of L and R and $R[j]$ is the smallest element of R not yet copied to A. When k is incremented, the loop invariant remains true: $A[p..k-1]$ contains the $k-p$ smallest elements of $A[p..r]$in sorted order. $L[i]$and $R[j]$are the smallest elements in the respective arrays that are not in $A[p..k-1]$.
The case $L[i] \leq R[j]$ is similar.

Termination

On termination, $k = r + 1$. The loop invariant implies that $A\,[p..k] = A\,[p..r]$contains the smallest $k - p = r - p + 1$ elements of L and R in sorted order. These are all the non-sentinel elements of L and R, so has $A[p..r]$ been sorted, as required.

Recursive Correctness

To check a recursive algorithm for correctness, verify that

- recursive calls eventually terminate in base case(s)
- the base case(s) are handled correctly
- assuming that the recursive calls return correctly, the function returns correctly

Progress to Base

The base case for $\textsc{Merge-Sort}$ is $p = r$ corresponding to an array of length 1. We will show progress to the base case by showing that, in the recursive calls to (A, p, q) and (A, $q+1$, r), the lengths of $A\left[p..q\right]$ and $A\left[q+1..r\right]$ are positive and less than the length of $A\left[p..r\right]$.
If $p < r$, Claim that if $p < r$, then
$p \le q = \left\lfloor (p+r)/2 \right\rfloor < r$.

Note that $p \le r - 1$, so $p \le \left\lfloor (p+p)/2 \right\rfloor \le \left\lfloor (p+r)/2 \right\rfloor = q \le \left\lfloor (r-1+r)/2 \right\rfloor = r - 1 < r$, as required.

Base Case Correct

If $p = r$, the function call returns with no changes to A. This is correct because a subarray of length 1 is already sorted.

Return Correct

If $\textsc{Merge-Sort}(A, p, q)$ returns with $A[p..q]$ sorted and $\textsc{Merge-Sort}(A, q+1, r)$ returns with $A[q+1..r]$ sorted, then the $\textsc{Merge}(A, p, q, r)$ call returns with $A[p..r]$ sorted, as required.

Stable Sort

Why use $L\left[i\right] \leq R\left[j\right]$?

A sorting algorithm is said to be *stable* if items with equal keys remain in the same relative order after sorting.

If you sort an array of names alphabetically by first name, then alphabetically by last name using a stable sort, the array will be sorted by last name. Within a range of the array with the same last name, the array is sorted by first name.

T(n) for Merge

The MERGE routine runs in $\Theta(n)$ time, where n is the length of the array to be sorted. To see this, note that the loop test in the for-loop on k runs $r - p + 1 = n$ times, while all of the remaining lines run at most n times.

T(n) for Merge Sort

For simplicity, assume $n = 2^k$ for some k.
Approximately, we have

$$T(n) = \begin{cases} c & n = 1 \\ 2T\left(\frac{n}{2}\right) + cn & n \geq 2 \end{cases}$$

The cn term comes from the time taken by the $\textsc{Merge}$ call.
(We actually take c to be small to get a lower bound, and c to be large to get an upper bound on $T(n)$.)

Resolving T(n)

$$
\begin{aligned}
&T(n) = 2T\left(\tfrac{n}{2}\right) + cn \\
&= 2\left(2T\left(\tfrac{n}{2}/2\right) + c\tfrac{n}{2}\right) + cn = 4T\left(\tfrac{n}{4}\right) + 2cn \\
&= 4\left(2T\left(\tfrac{n}{8}\right) + c\tfrac{n}{4}\right) + 2cn = 8T\left(\tfrac{n}{8}\right) + 3cn \\
&\vdots \\
&= 2^k T\left(\tfrac{n}{2^k}\right) + kcn = cn + cn\log_2 n
\end{aligned}
$$

Conclude that $T(n)$ is $\Theta(n \log_2 n)$.

This calculation can also be organized with a tree structure.