Chapter 2

Michelle Bodnar, Andrew Lohr

April 12, 2016

Exercise 2.1-1

31	41	59	26	41	58
31	41	59	26	41	58
31	41	59	26	41	58
26	31	41	59	41	58
26	31	41	41	59	58
26	31	41	41	58	59

Exercise 2.1-2

Algorithm 1 Non-increasing Insertion-Sort(A)

```
1: for $j = 2$ to $A.length$ do
2:     $key = A[j]$
3:     // Insert $A[j]$ into the sorted sequence $A[1..j-1]$.
4:     $i = j - 1$
5:     while $i > 0$ and $A[i] < key$ do
6:         $A[i+1] = A[i]$
7:         $i = i - 1$
8:     end while
9: end for
10: $A[i+1] = key$
```

Exercise 2.1-3

On each iteration of the loop body, the invariant upon entering is that there is no index $k < j$ so that $A[k] = v$. In order to proceed to the next iteration of the loop, we need that for the current value of j, we do not have $A[j] = v$. If the loop is exited by line 5, then we have just placed an acceptable value in i on the previous line. If the loop is exited by exhausting all possible values of j, then we know that there is no index that has value j, and so leaving NIL in i is correct.

```
$i = NIL$
for $j = 1$ to $A.length$ do
    if $A[j] = v$ then
        $i = j$
        return $i$
    end if
    return $i$
end for
```

Exercise 2.1-4

Input: two n-element arrays A and B containing the binary digits of two numbers a and b.
Output: an $(n+1)$-element array C containing the binary digits of $a+b$.

Algorithm 2 Adding n-bit Binary Integers

```
carry = 0
for i=1 to n do
    $C[i] = (A[i] + B[i] + carry) \pmod 2$
    if $A[i] + B[i] + carry \geq 2$ then
        carry = 1
    else
        carry = 0
    end if
end for
C[n+1] = carry
```

Exercise 2.2-1

$n^3/1000 - 100n^2 - 100n + 3 \in \Theta(n^3)$

Exercise 2.2-2

Input: An n-element array A.
Output: The array A with its elements rearranged into increasing order.
The loop invariant of selection sort is as follows: At each iteration of the for loop of lines 1 through 10, the subarray $A[1..i-1]$ contains the $i-1$ smallest elements of A in increasing order. After $n-1$ iterations of the loop, the $n-1$ smallest elements of A are in the first $n-1$ positions of A in increasing order, so the n^{th} element is necessarily the largest element. Therefore we do not need to run the loop a final time. The best-case and worst-case running times of selection sort are $\Theta(n^2)$. This is because regardless of how the elements are initially arranged, on the i^{th} iteration of the main for loop the algorithm always inspects each of the remaining $n-i$ elements to find the smallest one remaining.

Algorithm 3 Selection Sort

```
for $i = 1$ to $n-1$ do
    $min = i$
    for $j = i+1$ to $n$ do
        // Find the index of the $i^{th}$ smallest element
        if $A[j] < A[min]$ then
            $min = j$
        end if
    end for
    Swap $A[min]$ and $A[i]$
end for
```

This yields a running time of

$$\sum_{i=1}^{n-1} n - i = n(n-1) - \sum_{i=1}^{n-1} i = n^2 - n - \frac{n^2-n}{2} = \frac{n^2-n}{2} = \Theta(n^2).$$

Exercise 2.2-3

Suppose that every entry has probability p of being the element looked for. Then, we will only check k elements if the previous $k-1$ positions were not the element being looked for, and the kth position is the desired value. Taking the expected value of this distribution we get it to be

$$(1-p)^{A.length} + \sum_{k=1}^{A.length} k(1-p)^{k-1}p^k$$

Exercise 2.2-4

For a good best-case running time, modify an algorithm to first randomly produce output and then check whether or not it satisfies the goal of the algorithm. If so, produce this output and halt. Otherwise, run the algorithm as usual. It is unlikely that this will be successful, but in the best-case the running time will only be as long as it takes to check a solution. For example, we could modify selection sort to first randomly permute the elements of A, then check if they are in sorted order. If they are, output A. Otherwise run selection sort as usual. In the best case, this modified algorithm will have running time $\Theta(n)$.

Exercise 2.3-1

If we start with reading across the bottom of the tree and then go up level by level.

3	41	52	26	38	57	9	49
3	41	26	52	38	57	9	49
3	26	41	52	9	38	49	57
3	9	26	38	41	49	52	57

Exercise 2.3-2

The following is a rewrite of MERGE which avoids the use of sentinels. Much like MERGE, it begins by copying the subarrays of A to be merged into arrays L and R. At each iteration of the while loop starting on line 13 it selects the next smallest element from either L or R to place into A. It stops if either L or R runs out of elements, at which point it copies the remainder of the other subarray into the remaining spots of A.

Algorithm 4 $Merge(A, p, q, r)$

```
1: $n_1 = q - p + 1$
2: $n_2 = r - q$
3: let $L[1, ..n_1]$ and $R[1..n_2]$ be new arrays
4: **for** $i = 1$ to $n_1$ **do**
5:     $L[i] = A[p + i - 1]$
6: **end for**
7: **for** $j = 1$ to $n_2$ **do**
8:     $R[j] = A[q + j]$
9: **end for**
10: $i = 1$
11: $j = 1$
12: $k = p$
13: **while** $i \neq n_1 + 1$ and $j \neq n_2 + 1$ **do**
14:     **if** $L[i] \leq R[j]$ **then**
15:         $A[k] = L[i]$
16:         $i = i + 1$
17:     **else** $A[k] = R[j]$
18:         $j = j + 1$
19:     **end if**
20:     $k = k + 1$
21: **end while**
22: **if** $i == n_1 + 1$ **then**
23:     **for** $m = j$ to $n_2$ **do**
24:         $A[k] = R[m]$
25:         $k = k + 1$
26:     **end for**
27: **end if**
28: **if** $j == n_2 + 1$ **then**
29:     **for** $m = i$ to $n_1$ **do**
30:         $A[k] = L[m]$
31:         $k = k + 1$
32:     **end for**
33: **end if**
```

Exercise 2.3-3

Since n is a power of two, we may write $n = 2^k$. If $k = 1$, $T(2) = 2 = 2\lg(2)$. Suppose it is true for k, we will show it is true for $k+1$.

$$T(2^{k+1}) = 2T\left(\frac{2^{k+1}}{2}\right) + 2^{k+1} = 2T\left(2^k\right) + 2^{k+1} = 2(2^k \lg(2^k)) + 2^{k+1}$$

$$= k2^{k+1} + 2^{k+1} = (k+1)2^{k+1} = 2^{k+1}\lg(2^{k+1}) = n\lg(n)$$

Exercise 2.3-4

Let $T(n)$ denote the running time for insertion sort called on an array of size n. We can express $T(n)$ recursively as

$$T(n) = \begin{cases} \Theta(1) & \text{if } n \le c \\ T(n-1)I(n) & \text{otherwise} \end{cases}$$

where $I(n)$ denotes the amount of time it takes to insert $A[n]$ into the sorted array $A[1..n-1]$. As seen in exercise 2.3-5, $I(n)$ is $\Theta(\log n)$.

Exercise 2.3-5

The following recursive algorithm gives the desired result when called with $a = 1$ and $b = n$.

```
BinSearch(a,b,v)
if then$a > b$
    return NIL
end if
$m = \lfloor \frac{a+b}{2} \rfloor$
if then$m = v$
    return $m$
end if
if then$m < v$
    return BinSearch(a,m,v)
end if
return BinSearch(m+1,b,v)
```

Note that the initial call should be $BinSearch(1, n, v)$. Each call results in a constant number of operations plus a call to a problem instance where the quantity $b - a$ falls by at least a factor of two. So, the runtime satisfies the recurrence $T(n) = T(n/2) + c$. So, $T(n) \in \Theta(\lg(n))$

Exercise 2.3-6

A binary search wouldn't improve the worst-case running time. Insertion sort has to copy each element greater than *key* into its neighboring spot in the array. Doing a binary search would tell us how many how many elements need to be copied over, but wouldn't rid us of the copying needed to be done.

Exercise 2.3-7

```
 1: Use Merge Sort to sort the array $A$ in time $\Theta(n \lg(n))$
 2: $i = 1$
 3: $j = n$
 4: while $i < j$ do
 5:     if $A[j] + A[j] = S$ then
 6:         return true
 7:     end if
 8:     if $A[i] + A[j] < S$ then
 9:         $i = i + 1$
10:     end if
11:     if $A[i] + A[j] > S$ then
12:         $j = j - 1$
13:     end if
14: end while
15: return false
```

We can see that the while loop gets run at most $O(n)$ times, as the quantity $j-i$ starts at $n-1$ and decreases at each step. Also, since the body only consists of a constant amount of work, all of lines 2-15 takes only $O(n)$ time. So, the runtime is dominated by the time to perform the sort, which is $\Theta(n \lg(n))$. We will prove correctness by a mutual induction. Let $m_{i,j}$ be the proposition $A[i] + A[j] < S$ and $M_{i,j}$ be the proposition $A[i] + A[j] > S$. Note that because the array is sorted, $m_{i,j} \Rightarrow \forall k < j, m_{i,k}$, and $M_{i,j} \Rightarrow \forall k > i, M_{k,j}$.

Our program will obviously only output true in the case that there is a valid i and j. Now, suppose that our program output false, even though there were some i, j that was not considered for which $A[i] + A[j] = S$. If we have $i > j$, then swap the two, and the sum will not change, so, assume $i \leq j$. we now have two cases:

Case 1 $\exists k, (i, k)$ was considered and $j < k$. In this case, we take the smallest such k. The fact that this is nonzero meant that immediately after considering it, we considered (i+1,k) which means $m_{i,k}$ this means $m_{i,j}$

Case 2 $\exists k, (k, j)$ was considered and $k < i$. In this case, we take the largest such k. The fact that this is nonzero meant that immediately after considering it, we considered (k,j-1) which means $M_{k,j}$ this means $M_{i,j}$

Note that one of these two cases must be true since the set of considered points separates $\{(m, m') : m \leq m' < n\}$ into at most two regions. If you are in the region that contains $(1, 1)$(if nonempty) then you are in Case 1. If you are in the region that contains (n, n) (if non-empty) then you are in case 2.

Problem 2-1

a. The time for insertion sort to sort a single list of length k is $\Theta(k^2)$, so, n/k of them will take time $\Theta(\frac{n}{k}k^2) = \Theta(nk)$.

b. Suppose we have coarseness k. This meas we can just start using the usual merging procedure, except starting it at the level in which each array has size at most k. This means that the depth of the merge tree is $\lg(n) - \lg(k) = \lg(n/k)$. Each level of merging is still time cn, so putting it together, the merging takes time $\Theta(n \lg(n/k))$.

c. Viewing k as a function of n, as long as $k(n) \in O(\lg(n))$, it has the same asymptotics. In particular, for any constant choice of k, the asymptotics are the same.

d. If we optimize the previous expression using our calculus 1 skills to get k, we have that $c_1 n - \frac{nc_2}{k} = 0$ where c_1 and c_2 are the coeffients of nk and $n \lg(n/k)$ hidden by the asymptotics notation. In particular, a constant choice of k is optimal. In practice we could find the best choice of this k by just trying and timing for various values for sufficiently large n.

Problem 2-2

1. We need to prove that A' contains the same elements as A, which is easily seen to be true because the only modification we make to A is swapping its elements, so the resulting array must contain a rearrangement of the elements in the original array.

2. The **for** loop in lines 2 through 4 maintains the following loop invariant: At the start of each iteration, the position of the smallest element of $A[i..n]$ is at most j. This is clearly true prior to the first iteration because the position of any element is at most $A.length$. To see that each iteration maintains the loop invariant, suppose that $j = k$ and the position of the smallest element of $A[i..n]$ is at most k. Then we compare $A[k]$ to $A[k-1]$. If $A[k] < A[k-1]$ then $A[k-1]$ is not the smallest element of $A[i..n]$, so when we swap $A[k]$ and $A[k-1]$ we know that the smallest element of $A[i..n]$ must occur in the first $k-1$ positions of the subarray, the maintaining the invariant. On the other hand, if $A[k] \geq A[k-1]$ then the smallest element can't be $A[k]$. Since we do nothing, we conclude that the smallest element has position at most $k-1$. Upon termination, the smallest element of $A[i..n]$ is in position i.

3. The **for** loop in lines 1 through 4 maintain the following loop invariant: At the start of each iteration the subarray $A[1..i-1]$ contains the $i-1$ smallest elements of A in sorted order. Prior to the first iteration $i = 1$, and the first 0 elements of A are trivially sorted. To see that each iteration maintains the loop invariant, fix i and suppose that $A[1..i-1]$ contains the $i-1$ smallest elements of A in sorted order. Then we run the loop in lines 2 through 4. We showed in part b that when this loop terminates, the smallest element of $A[i..n]$ is in position i. Since the $i-1$ smallest elements of A are already in $A[1..i-1]$, $A[i]$ must be the i^{th} smallest element of A. Therefore $A[1..i]$ contains the i smallest elements of A in sorted order, maintaining the loop invariant. Upon termination, $A[1..n]$ contains the n elements of A in sorted order as desired.

4. The i^{th} iteration of the **for** loop of lines 1 through 4 will cause $n-i$ iterations of the **for** loop of lines 2 through 4, each with constant time execution, so the worst-case running time is $\Theta(n^2)$. This is the same as that of insertion sort; however, bubble sort also has best-case running time $\Theta(n^2)$ whereas insertion sort has best-case running time $\Theta(n)$.

Problem 2-3

a. If we assume that the arithmetic can all be done in constant time, then since the loop is being executed n times, it has runtime $\Theta(n)$.

b.
```
1: $y = 0$
2: for i=0 to n do
3:     $y_i = x$
4:     for j=1 to n do
5:         $y_i = y_i x$
6:     end for
7:     $y = y + a_i y_i$
8: end for
```

This code has runtime $\Theta(n^2)$ because it has to compute each of the powers of x. This is slower than Horner's rule.

c. Initially, $i = n$, so, the upper bound of the summation is -1, so the sum evaluates to 0, which is the value of y. For preservation, suppose it is true for an i, then,

$$y = a_i + x \sum_{k=0}^{n-(i+1)} a_{k+i+1} x^k = a_i + x \sum_{k=1}^{n-i} a_{k+i} x^{k-1} = \sum_{k=0}^{n-i} a_{k+i} x^k$$

At termination, $i = 0$, so is summing up to $n-1$, so executing the body of the loop a last time gets us the desired final result.

d. We just showed that the algorithm evaluated $\Sigma_{k=0}^{n} a_k x^k$. This is the value of the polynomial evaluated at x.

Problem 2-4

a. The five inversions are $(2, 1)$, $(3, 1)$, $(8, 6)$, $(8, 1)$, and $(6, 1)$.

b. The n-element array with the most inversions is $\langle n, n-1, \ldots, 2, 1\rangle$. It has $n - 1 + n - 2 + \ldots + 2 + 1 = \frac{n(n-1)}{2}$ inversions.

c. The running time of insertion sort is a constant times the number of inversions. Let $I(i)$ denote the number of $j < i$ such that $A[j] > A[i]$. Then $\sum_{i=1}^{n} I(i)$ equals the number of inversions in A. Now consider the **while** loop on lines 5-7 of the insertion sort algorithm. The loop will execute once for each element of A which has index less than j is larger than $A[j]$. Thus, it will execute $I(j)$ times. We reach this **while** loop once for each iteration of the **for** loop, so the number of constant time steps of insertion sort is $\sum_{j=1}^{n} I(j)$ which is exactly the inversion number of A.

d. We'll call our algorithm M.Merge-Sort for Modified Merge Sort. In addition to sorting A, it will also keep track of the number of inversions. The algorithm works as follows. When we call M.Merge-Sort(A,p,q) it sorts $A[p..q]$ and returns the number of inversions in the elements of $A[p..q]$, so $left$ and $right$ track the number of inversions of the form (i, j) where i and j are both in the same half of A. When M.Merge(A,p,q,r) is called, it returns the number of inversions of the form (i, j) where i is in the first half of the array and j is in the second half. Summing these up gives the total number of inversions in A. The runtime is the same as that of Merge-Sort because we only add an additional constant-time operation to some of the iterations of some of the loops. Since Merge is $\Theta(n \log n)$, so is this algorithm.

Algorithm 5 M.Merge-Sort(A, p, r)

```
if $p < r$ then
    $q = \lfloor (p + r)/2 \rfloor$
    $left = M.Merge - Sort(A, p, q)$
    $right = M.Merge - Sort(A, q + 1, r)$
    $inv = M.Merge(A, p, q, r) + left + right$
    return $inv$
end if
return 0
```

Algorithm 6 M.Merge(A,p,q,r)

```
inv = 0
$n_1 = q - p + 1$
$n_2 = r - q$
let $L[1, ..n_1]$ and $R[1..n_2]$ be new arrays
for $i = 1$ to $n_1$ do
    $L[i] = A[p + i - 1]$
end for
for $j = 1$ to $n_2$ do
    $R[j] = A[q + j]$
end for
$i = 1$
$j = 1$
$k = p$
while $i \neq n_1 + 1$ and $j \neq n_2 + 1$ do
    if $L[i] \leq R[j]$ then
        $A[k] = L[i]$
        $i = i + 1$
    else $A[k] = R[j]$
        $inv = inv + j$ // This keeps track of the number of inversions between the left and right arrays.
        $j = j + 1$
    end if
    $k = k + 1$
end while
if $i == n_1 + 1$ then
    for $m = j$ to $n_2$ do
        $A[k] = R[m]$
        $k = k + 1$
    end for
end if
if $j == n_2 + 1$ then
    for $m = i$ to $n_1$ do
        $A[k] = L[m]$
        $inv = inv + n_2$ // Tracks inversions once we have exhausted the right array. At this point, every element of the right array contributes an inversion.
        $k = k + 1$
    end for
end if
return inv
```