

2.2-1

Express the function $n^3/1000 - 100n^2 - 100n + 3$ in terms of Θ -notation.

Solution.

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

2.3-2

Rewrite the MERGE procedure so that it does not use sentinels, instead stopping once either array L or R has had all its elements copied back to A and then copying the remainder of the other array back into A .

Solution.

```
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  $n_1$ 
5       $L[i] = A[p + i - 1]$ 
6  for  $j = 1$  to  $n_2$ 
7       $R[j] = A[q + j]$ 
8   $i = 1$ 
9   $j = 1$ 
10 for  $k = p$  to  $r$ 
11     if  $j > n_2$  or ( $i \leq n_1$  and  $L[i] \leq R[j]$ )
12          $A[k] = L[i]$ 
13          $i = i + 1$ 
14     else  $A[k] = R[j]$ 
15          $j = j + 1$ 
```

2.3-3

Use mathematical induction to show that when n is an exact power of 2, the solution of the recurrence

$$T(n) = \begin{cases} 2 & \text{if } n = 2, \\ 2T(n/2) + n & \text{if } n = 2^k, \text{ for } k > 1 \end{cases}$$

is $T(n) = n \lg n$.

Solution.

Induction Basis: When $n = 2$, $T(n) = n \lg n$ holds.

Inductive Step: Suppose that $T(n) = n \lg n$ holds for $n = 2^k$, i.e. $T(2^k) = 2^k \lg 2^k = k \cdot 2^k$. Then $T(2^{k+1}) = 2T(2^k) + 2^{k+1} = k \cdot 2^{k+1} + 2^{k+1} = (k+1) \cdot 2^{k+1} = 2^{k+1} \lg 2^{k+1}$, i.e. $T(n) = n \lg n$ holds for $n = 2^{k+1}$.

2.3-4

We can express insertion sort as a recursive procedure as follows. In order to sort $A[1..n]$, we recursively sort $A[1..n-1]$ and then insert $A[n]$ into the sorted array $A[1..n-1]$. Write a recurrence for the running time of this recursive version of insertion sort.

Solution.

$$T(n) = \begin{cases} 0 & \text{if } n = 1 \\ T(n-1) + c_1n + c_2 & \text{if } n \geq 2 \end{cases}$$

2.3-5

Referring back to the searching algorithm (see Exercise 2.1-3), observe that if the sequence A is sorted, we can check the midpoint of the sequence against v and eliminate half of the sequence from further consideration. The **binary search** algorithm repeats this procedure, halving the size of the remaining portion of the sequence each time. Write pseudocode, either iterative or recursive, for binary search. Argue that the worst-case running time of binary search is $\Theta(\lg n)$.

Solution.

The following procedure takes an array A , its size n and the element to find x , and returns the index of x within A . If x is not found in A , a special value, NOT-FOUND, is returned. The loop invariant is: If x can be found in A , it is in the subarray $A[l..h]$.

BINARY-SEARCH(A, n, x)

```

1   $l = 1$ 
2   $h = n$ 
3  while  $l \leq h$ 
4       $m = \lfloor (l + h)/2 \rfloor$ 
5      if  $A[m] == x$ 
6          return  $m$ 
7      elseif  $A[m] < x$ 
8           $l = m + 1$ 
9      else  $h = m - 1$ 
10 return NOT-FOUND
```

The running time is linear to the number of iterations of the loop in lines 3-9. In the worst case, element x is not found in A . Each iteration reduces the size of the subarray in half¹, so the loop runs $\Theta(\lg n)$ iterations. Hence the worst-case running time is $\Theta(\lg n)$.

2.3-6

Observe that the **while** loop of lines 5-7 of the INSERTION-SORT procedure in Section 2.1 uses a linear search to scan (backward) through the sorted subarray

¹We are being sloppy here. Actually, each iteration reduces a subarray of size m to either $\lfloor (m-1)/2 \rfloor$ or $\lceil (m-1)/2 \rceil$, neither being exactly $m/2$. A rigorous argument will require more details.

$A[1 \dots j - 1]$. Can we use a binary search (see Exercise 2.3-5) instead to improve the overall worst-case running time of insertion sort to $\Theta(n \lg n)$?

Solution.

No. Although we can now *find* the point to insert $A[j]$ into the sorted subarray $A[1 \dots j - 1]$ in at most $\Theta(\lg n)$ time, in the worst case it will still take $\Theta(n)$ time to actually *insert* the element.