

Solution to Exercise 2.2-2

SELECTION-SORT(A)

$n \leftarrow \text{length}[A]$

for $j \leftarrow 1$ **to** $n - 1$

do $\text{smallest} \leftarrow j$

for $i \leftarrow j + 1$ **to** n

do if $A[i] < A[\text{smallest}]$

then $\text{smallest} \leftarrow i$

 exchange $A[j] \leftrightarrow A[\text{smallest}]$

The algorithm maintains the loop invariant that at the start of each iteration of the outer **for** loop, the subarray $A[1 \dots j - 1]$ consists of the $j - 1$ smallest elements in the array $A[1 \dots n]$, and this subarray is in sorted order. After the first $n - 1$ elements, the subarray $A[1 \dots n - 1]$ contains the smallest $n - 1$ elements, sorted, and therefore element $A[n]$ must be the largest element. The running time of the algorithm is $\Theta(n^2)$ for all cases.

Solution to Exercise 2.3-5

Procedure BINARY-SEARCH takes a sorted array A , a value v , and a range [$low \dots high$] of the array, in which we search for the value v . The procedure compares v to the array entry at the midpoint of the range and decides to eliminate half the range from further consideration. We give both iterative and recursive versions, each of which returns either an index i such that $A[i] = v$, or NIL if no entry of $A[low \dots high]$ contains the value v . The initial call to either version should have the parameters $A, v, 1, n$.

ITERATIVE-BINARY-SEARCH($A, v, low, high$)

while $low \leq high$

do $mid \leftarrow \lfloor (low + high) / 2 \rfloor$

if $v = A[mid]$

then return mid

if $v > A[mid]$

then $low \leftarrow mid + 1$

else $high \leftarrow mid - 1$

return NIL

RECURSIVE-BINARY-SEARCH($A, v, low, high$)

if $low > high$

```

    then return NIL
mid ← ⌊(low + high) / 2⌋
if v = A[mid]
    then return mid
if v > A[mid]
    then return RECURSIVE-BINARY-SEARCH(A, v, mid + 1, high)
    else return RECURSIVE-BINARY-SEARCH(A, v, low, mid - 1)

```

Both procedures terminate the search unsuccessfully when the range is empty (i.e., $low > high$) and terminate it successfully if the value v has been found. Based on the comparison of v to the middle element in the searched range, the search continues with the range halved. The recurrence for these procedures is therefore $T(n) = T(n/2) + \Theta(1)$, whose solution is $T(n) = \Theta(\lg n)$.

Solution to Exercise 2.3-7

The following algorithm solves the problem:

1. Sort the elements in S .
2. Form the set $S' = \{z : z = x - y \text{ for some } y \in S\}$.
3. Sort the elements in S' .
4. If any value in S appears more than once, remove all but one instance. Do the same for S' .
5. Merge the two sorted sets S and S' .
6. There exist two elements in S whose sum is exactly x if and only if the same value appears in consecutive positions in the merged output.

To justify the claim in step 4, first observe that if any value appears twice in the merged output, it must appear in consecutive positions. Thus, we can restate the condition in step 5 as there exist two elements in S whose sum is exactly x if and only if the same value appears twice in the merged output.

Suppose that some value w appears twice. Then w appeared once in S and once in S' . Because w appeared in S' , there exists some $y \in S$ such that $w = x - y$, or $x = w + y$. Since $w \in S$, the elements w and y are in S and sum to x .

Conversely, suppose that there are values $w, y \in S$ such that $w + y = x$. Then, since $x - y = w$, the value w appears in S' . Thus, w is in both S and S' , and so it will appear twice in the merged output.

Steps 1 and 3 require $\Theta(n \lg n)$ steps. Steps 2, 4, 5, and 6 require $\Theta(n)$ steps. Thus the overall running time is $\Theta(n \lg n)$.

Solution to Problem 2-1

a. Insertion sort takes $\Theta(k^2)$ time per k -element list in the worst case. Therefore, sorting n/k lists of k elements each takes $\Theta(k^2 n / k) = \Theta(nk)$ worst-case time.

b. Just extending the 2-list merge to merge all the lists at once would take $\Theta(n \cdot (n/k)) = \Theta(n^2/k)$ time (n from copying each element once into the result list, n/k from examining n/k lists at each step to select next item for result list). To achieve $\Theta(n \cdot \lg(n/k))$ -time merging, we merge the lists pairwise, then merge the resulting lists pairwise, and so on, until there's just one list. The pairwise merging requires $\Theta(n)$ work at each level, since we are still working on n elements, even if they are partitioned among sublists. The number of levels, starting with n/k lists (with k elements each) and finishing with 1 list (with n elements), is $\lceil \lg(n/k) \rceil$. Therefore, the total running time for the merging is $\Theta(n \cdot \lg(n/k))$.

c. The modified algorithm has the same asymptotic running time as standard merge sort when $\Theta(nk + n \cdot \lg(n/k)) = \Theta(n \cdot \lg n)$. The largest asymptotic value of k as a function of n that satisfies this condition is $k = \Theta(\lg n)$. To see why, first observe that k cannot be more than $\Theta(\lg n)$ (i.e., it can't have a higher-order term than $\lg n$), for otherwise the left-hand expression wouldn't be $\Theta(n \lg n)$ (because it would have a higher-order term than $n \lg n$). So all we need to do is verify that $k = \Theta(\lg n)$ works, which we can do by plugging $k = \lg n$ into $\Theta(nk + n \cdot \lg(n/k)) = \Theta(nk + n \lg n - n \lg k)$ to get $\Theta(n \lg n + n \lg n - n \lg \lg n) = \Theta(2n \lg n - n \lg \lg n)$, which, by taking just the high-order term and ignoring the constant coefficient, equals $\Theta(n \cdot \lg n)$.

d. In practice, k should be the largest list length on which insertion sort is faster than merge sort.

Solution to Exercise 3.1-4

$2^{n+1} = O(2^n)$, but $2^{2^n} \neq O(2^n)$.

To show that $2^{n+1} = O(2^n)$, we must find constants $c, n_0 > 0$ such that

$$0 \leq 2^{n+1} \leq c \cdot 2^n \text{ for all } n \geq n_0.$$

Since $2^{n+1} = 2 \cdot 2^n$ for all n , we can satisfy the definition with $c = 2$ and $n_0 = 1$.

To show that $2^{2^n} \neq O(2^n)$, assume there exist constants $c, n_0 > 0$ such that

$$0 \leq 2^{2^n} \leq c \cdot 2^n \text{ for all } n \geq n_0.$$

Then $2^{2^n} = 2^n \cdot 2^n \leq c \cdot 2^n \Rightarrow 2^n \leq c$. But no constant is greater than all 2^n , and so the assumption leads to a contradiction.

Solution to Exercise 3.2-4

$\lceil \lg n \rceil!$ is not polynomially bounded, but $\lceil \lg \lg n \rceil!$ is.

Proving that a function $f(n)$ is polynomially bounded is equivalent to proving that

$\lg(f(n)) = O(\lg n)$ for the following reasons.

- If f is polynomially bounded, then there exist constants c, k, n_0 such that for all $n \geq n_0$, $f(n) \leq cn^k$. Hence, $\lg(f(n)) \leq k \cdot c \cdot \lg n$, which, since c and k are constants, means that $\lg(f(n)) = O(\lg n)$.
- Similarly, if $\lg(f(n)) = O(\lg n)$, then f is polynomially bounded.

In the following proofs, we will make use of the following two facts:

1. $\lg(n!) = \Theta(n \lg n)$ (by equation (3.18)).

2. $\lceil \lg n \rceil = \Theta(\lg n)$, because $\lceil \lg n \rceil \geq \lg n$ and $\lceil \lg n \rceil < \lg n + 1 \leq 2 \lg n$ for all $n \geq 2$

$$\begin{aligned} \lg(\lceil \lg n \rceil!) &= \Theta(\lceil \lg n \rceil \lg \lceil \lg n \rceil) \\ &= \Theta(\lg n \lg \lg n) \\ &= \omega(\lg n). \end{aligned}$$

Therefore, $\lg(\lceil \lg n \rceil!) \neq O(\lg n)$, and so $\lceil \lg n \rceil!$ is not polynomially bounded.

$$\begin{aligned} \lg(\lceil \lg \lg n \rceil!) &= \Theta(\lceil \lg \lg n \rceil \lg \lceil \lg \lg n \rceil) \\ &= \Theta(\lg \lg n \lg \lg \lg n) \\ &= o((\lg \lg n)^2) \\ &= o(\lg^2(\lg n)) \\ &= o(\lg n). \end{aligned}$$

The last step above follows from the property that any polylogarithmic function grows more slowly than any positive polynomial function, i.e., that for constants $a, b > 0$, we have

$\lg^b(\lg n) = o(n^a)$. Substitute $\lg n$ for n , 2 for b , and 1 for a , giving $\lg^2(\lg n) = o(\lg n)$.

Therefore, $\lg(\lceil \lg \lg n \rceil!) = O(\lg n)$, and so $\lceil \lg \lg n \rceil!$ is polynomially bounded.