2.12. Let $L(n)$ be the number of lines. Then $L(n) = 2L(n/2) + 1$ if $n > 1$ and $L(1) = 1$. Solving, we get $L(n) = 2n - 1 = \Theta(n)$.

2.16. pos = 1
while $A[pos] < x$:
    pos = 2 * pos
return binary-search($A[1..pos]$, $x$)

The algorithm first finds an index with value greater than or equal to $x$, then invokes a standard binary search procedure. During the initial phase, $pos$ is doubled at most $\log n$ times (because after this many doublings, it exceeds $n$, and thus $A[pos] > x$); thus the overall running time is $O(\log n)$.

2.19. (a) The $merge$ procedure takes time $O(k + l)$ to merge two arrays of size $k$ and $l$. In this case, we are performing $k - 1$ merge operations. In the $j$th such operation, one of the arrays has size $jn$ while the other has size $n$; therefore the time taken for the operation is $O((j + 1)n)$. The total time taken is then $O(2n + 3n + 4n + \ldots + kn) = O(k^2n)$.

(b) Let $multimerge(A_1, A_2, \ldots, A_k)$ denote the result of merging arrays $A_1, \ldots, A_k$, and let $merge$ be our two-way merge operation from class. We can do a multimerge using divide and conquer:

$$multimerge(A_1, A_2, \ldots, A_k) = merge(multimerge(A_1, A_2, \ldots, A_{\lfloor k/2 \rfloor}), multimerge(A_{\lfloor k/2 \rfloor + 1}, \ldots, A_k)).$$

Let $T(k)$ denote the time taken to multimerge $k$ arrays of length $n$. Then

$$T(k) = 2T(k/2) + O(kn)$$

because there are two recursive calls to $multimerge$, each taking time $T(k/2)$, and the final $merge$ operation involves $kn$ elements. The solution to this recurrence is $T(k) = nk \log k$, a big improvement over the scheme in part (a).

2.22. Here’s the algorithm (assuming for convenience that $m, n$ are powers of two).

```python
function getelement(x[1...n], y[1...m], k)
if n = 0:    return y[k]
if m = 0:    return x[k]
if x[n/2] > y[m/2]:
    if k < (m + n)/2:
        return getelement(x[1...n/2], y[1...m], k)
    else:
        return getelement(x[1...n], y[(m/2) + 1...m], k - m/2)
else:
    if k < (m + n)/2:
        return getelement(x[1...n], y[1...m/2], k)
    else:
        return getelement(x[(n/2) + 1...n], y[1...m], k - n/2)
```

**Brief justification:** If $x[n/2] > y[m/2]$, then the top half of array $x$ is greater than the bottom halves of both arrays. Therefore, the entire top half of array $x$ must lie above the median of the combined arrays. Similarly, the entire bottom half of array $y$ must lie below the median of the combined arrays. By comparing $k$ to $(m + n)/2$, we can therefore eliminate one of these two half-arrays. The other cases are similar.


**Running time:** In each recursive call, a constant amount of time is taken and either \( m \) or \( n \) gets halved in value. This can happen at most \( \log m + \log n \) times before one of them reaches zero. Therefore the total running time is \( O(\log m + \log n) \).

2.23. (a) **Solving the problem in \( O(n \log n) \) time.**

Here’s a divide-and-conquer algorithm:

```python
function majority (A[1...n])
    if n = 1: return A[1]
    let A_L, A_R be the first and second halves of A
    M_L = majority(A_L) and M_R = majority(A_R)
    if neither half has a majority:
        return ‘no majority’
    else:
        check whether either M_L or M_R is a majority element of A
        if so, return that element; else return ‘no majority’
```

**Brief justification:** If \( A \) has a majority element \( x \), then \( x \) appears more than \( n/2 \) times in \( A \) and thus appears more than \( n/4 \) times in either \( A_L \) or \( A_R \); it follows that \( x \) must also be a majority element of one (or both) of these two arrays.

**Running time:** \( T(n) = 2T(n/2) + O(n) = O(n \log n) \).

(b) **A linear-time algorithm.**

```python
function majority (A[1...n])
    x = prune(A)
    if x is a majority element of A:
        return x
    else:
        return ‘no majority’

function prune (S[1...n])
    if n = 1: return S[1]
    S' = [ ] (empty list)
    for i = 1 to n/2:
        if S[2i - 1] = S[2i]: add S[2i] to S'
    return prune(S')
```

**Justification:** We’ll show that each iteration of the `prune` procedure maintains the following invariant: if \( x \) is a majority element of \( S \) then it is also a majority element of \( S' \). The rest then follows.

Suppose \( x \) is a majority element of \( S \). In an iteration of `prune`, we break \( S \) into pairs. Suppose there are \( k \) pairs of Type One and \( l \) pairs of Type Two:

- **Type One:** the two elements are different. In this case, we discard both.
- **Type Two:** the elements are the same. In this case, we keep one of them.

Since \( x \) constitutes at most half of the elements in the Type One pairs, \( x \) must be a majority element in the Type Two pairs. At the end of the iteration, what remains are \( l \) elements, one from each Type Two pair. Therefore \( x \) is the majority of these elements.

**Running time.** In each iteration of `prune`, the number of elements in \( S \) is reduced to \( l \leq |S|/2 \). Therefore, the total time taken is \( T(n) \leq T(n/2) + O(n) = O(n) \).