the majority must be the median, we can scan the sequence to test if the median is indeed the majority. This method takes $\Theta(n)$ time, as the median can be found in $\Theta(n)$ time. As we will see in Sec. 6.5, the hidden constant in the time complexity of the median finding algorithm is too large, and the algorithm is fairly complex.

It turns out that there is an elegant solution that uses much fewer comparisons. We derive this algorithm using induction. The essence of the algorithm is based on the following observation:

**Observation 5.1** If two different elements in the original sequence are removed, then the majority in the original sequence remains the majority in the new sequence.

This observation suggests the following procedure for finding an element that is a candidate for being the majority. Set a counter to zero and let $x = A[1]$. Starting from $A[2]$, scan the elements one by one increasing the counter by one if the current element is equal to $x$ and decreasing the counter by one if the current element is not equal to $x$. If all the elements have been scanned and the counter is greater than zero, then return $x$ as the candidate. If the counter becomes zero when comparing $x$ with $A[j]$, $1 < j < n$, then call procedure candidate recursively on the elements $A[j+1..n]$. Notice that decrementing the counter implements the idea of throwing two different elements as stated in Observation 5.1. This method is described more precisely in Algorithm majority. Converting this recursive algorithm into an iterative one is straightforward, and is left as an exercise.

### 5.8 Exercises

5.1. Give a recursive algorithm that computes the $n$th Fibonacci number $f_n$ defined by

$$f_1 = f_2 = 1; \quad f_n = f_{n-1} + f_{n-2} \text{ for } n \geq 3.$$  

5.2. Give an iterative algorithm that computes the $n$th Fibonacci number $f_n$ defined above.

5.3. Use induction to develop a recursive algorithm for finding the maximum element in a given sequence $A[1..n]$ of $n$ elements.
Algorithm 5.9 MAJORITY
Output: The majority element if it exists; otherwise none.

1. $c ← \text{candidate}(1)$
2. $count ← 0$
3. for $j ← 1$ to $n$
   4. if $A[j] = c$ then $count ← count + 1$
5. end for
6. if $count > \lceil n/2 \rceil$ then return $c$
7. else return none

Procedure candidate($m$)
1. $j ← m; c ← A[m]; count ← 1$
2. while $j < n$ and count > 0
3. $j ← j + 1$
4. if $A[j] = c$ then $count ← count + 1$
5. else $count ← count - 1$
6. end while
7. if $j = n$ then return $c$ {See Exercises 5.31 and 5.32.}
8. else return candidate($j + 1$)

5.4. Use induction to develop a recursive algorithm for finding the average of $n$ real numbers $A[1..n]$.

5.5. Use induction to develop a recursive algorithm that searches for an element $x$ in a given sequence $A[1..n]$ of $n$ elements.

5.6. Derive the running time of Algorithm INSERTIONSORTREC.

5.7. Illustrate the operation of Algorithm RADIXSORT on the following sequence of eight numbers:
   (a) 4567, 2463, 6523, 7461, 4251, 3241, 6492, 7563.
   (b) 16543, 25895, 18674, 98256, 91428, 73234, 16597, 73195.

5.8. Express the time complexity of Algorithm RADIXSORT in terms of $n$ when the input consists of $n$ positive integers in the interval
   (a) $[1..n]$.
   (b) $[1..n^2]$.
   (c) $[1..2^n]$.

5.9. Let $A[1..n]$ be an array of positive integers in the interval $[1..n!]$. Which sorting algorithm do you think is faster: BOTTOMUPSORT or RADIXSORT? {See Sec. 1.7.}

5.10. What is the time complexity of Algorithm RADIXSORT if arrays are used instead of linked lists? Explain.
5.11. Give a recursive version of Algorithm BUBBLESORT given in Exercise 1.16.

5.12. A sorting method known as bucket sort works as follows. Let $A[1..n]$ be a sequence of $n$ numbers within a reasonable range, say all numbers are between 1 and $m$, where $m$ is not too large compared to $n$. The numbers are distributed into $k$ buckets, with the first bucket containing those numbers between 1 and $\lfloor m/k \rfloor$, the second bucket containing those numbers between $\lfloor m/k \rfloor + 1$ to $\lfloor 2m/k \rfloor$, and so on. The numbers in each bucket are then sorted using another sorting algorithm, say Algorithm INSERTIONSORT. Analyze the running time of the algorithm.

5.13. Instead of using another sorting algorithm in Exercises 5.12, design a recursive version of bucket sort that recursively sorts the numbers in each bucket. What is the major disadvantage of this recursive version?

5.14. A sorting algorithm is called stable if the order of equal elements is preserved after sorting. Which of the following sorting algorithms are stable?
   (a) SELECTIONSORT  (b) INSERTIONSORT  (c) BUBBLESORT
   (d) BOTTOMUPSORT  (e) HEAPSORT  (f) RADIXSORT.

5.15. Use induction to solve Exercise 3.7.

5.16. Use induction to solve Exercise 3.8.

5.17. Use Horner’s rule described in Sec. 5.5 to evaluate the following polynomials:
   (a) $3x^5 + 2x^4 + 4x^3 + x^2 + 2x + 5$.
   (b) $2x^7 + 3x^5 + 2x^3 + 5x^2 + 3x + 7$.

5.18. Use Algorithm EXPRREC to compute
   (a) $2^5$.  (b) $2^7$.  (c) $3^5$.  (d) $5^7$.

5.19. Solve Exercise 5.18 using Algorithm EXP instead of Algorithm EXPRREC.

5.20. Carefully explain why in Algorithm PERMUTATIONS1 when $P[j]$ and $P[m]$ are interchanged before the recursive call, they must be interchanged back after the recursive call.

5.21. Carefully explain why in Algorithm PERMUTATIONS2 $P[j]$ must be reset to 0 after the recursive call.

5.22. Carefully explain why in Algorithm PERMUTATIONS2, when Procedure perm2 is invoked by the call perm2(m) with $m > 0$, the array $P$ contains exactly $m$ zeros, and hence the recursive call perm2($m - 1$) will be executed exactly $m$ times.

5.23. Modify Algorithm PERMUTATIONS2 so that the permutations of the numbers 1, 2, ..., $n$ are generated in a reverse order to that produced by Algorithm PERMUTATIONS2.
5.24. Modify Algorithm `permutations2` so that it generates all $k$-subsets of the set $\{1, 2, \ldots, n\}$, $1 \leq k \leq n$.

5.25. Analyze the time complexity of the modified algorithm in Exercise 5.24.


5.27. Prove the correctness of Algorithm `permutations2`.

5.28. Give an iterative version of Algorithm `majority`.

5.29. Illustrate the operation of Algorithm `majority` on the arrays

(a) \[5 \underline{7} 5 \underline{4} 5\]
(b) \[5 \underline{7} 5 \underline{4} 8\]
(c) \[2 \underline{4} 1 \underline{4} 4 6 4\]

5.30. Prove Observation 5.1.

5.31. Prove or disprove the following claim. If in Step 7 of Procedure `candidate` in Algorithm `majority` $j = n$ but $\text{count} = 0$ then $c$ is the majority element.

5.32. Prove or disprove the following claim. If in Step 7 of Procedure `candidate` in Algorithm `majority` $j = n$ and $\text{count} > 0$ then $c$ is the majority element.

5.33. Let $A[1..n]$ be a sorted array of $n$ integers, and $x$ an integer. Design an $O(n)$ time algorithm to determine whether there are two elements in $A$, if any, whose sum is exactly $x$.

5.9 Bibliographic notes

The use of induction as a mathematical technique for proving the correctness of algorithms was first developed by Floyd (1967). Recursion has been studied extensively in algorithm design. See for example the books of Burge (1975) and Paull (1988). The use of induction as a design technique appears in Manber (1988). Manber (1989) is a whole book that is mostly devoted to the induction design technique. Unlike this chapter, induction in that book encompasses a wide variety of problems and is used in its broad sense to cover other design techniques like divide and conquer and dynamic programming. Radix sort is used by card-sorting machines. In old machines, the machine did the distribution step and the operator collected the piles after each pass and combined them into one for the next pass. Horner's rule for polynomial evaluation is after the English mathematician W. G. Horner. Algorithm `permutations2` appears in Banachowski, Kreczmar