
Solution: The divide-and-conquer procedure `FindLocalMin(A[1..n])` finds a local minimum given $A[1..n]$ has the property described above. We’ll assume $n \geq 3$ so that at least one element of the array has two neighbors.

```plaintext
FindLocalMin(A[1..n]):
if n = 3
    return A[2]
else
    m ← ⌊n/2⌋
    if A[m] > A[m-1]
        return FindLocalMin(A[1..m])
    else if A[m] > A[m+1]
        return FindLocalMin(A[m+1..n])
    else
        return A[m]
```

If $n = 3$, then the special property of $A$ guarantees $A[2]$ is a local minimum by definition. Suppose $n > 3$. If $A[m] > A[m-1]$, then the subarray $A[1..m]$ has the special property needed for our algorithm. The Recursion Fairy finds a local minimum in this subarray during the first recursive call written in the code. Similarly, if $A[m] \leq A[m-1]$ but $A[m] > A[m+1]$, then the subarray $A[m..n]$ has the special property and the Recursion Fairy finds a local minimum during the second recursive call. If neither inequality holds, then $A[m]$ is a local minimum by definition, and we are safe to return it.

The algorithm performs at most one recursive call on an array of about half the input size and does a constant amount of work otherwise. The running time follows the recurrence $T(n) \leq T(n/2) + O(1)$. The applicable recursion tree has $\log_2 n$ levels all summing to the same constant so the running time is $T(n) = O(\log n)$. (Also, this recurrence is the standard one for binary search.)

Rubric: 5 points total: 3 points for the algorithm, 1 point for the justification, 1 point for the running time analysis. −0.5 points for forgetting the base case.

(b) Suppose we are given two sorted arrays $A[1..n]$ and $B[1..n]$. The median element in the union of $A$ and $B$ is the $n$th smallest element in their union. Describe and analyze an algorithm that finds this median element in $O(\log n)$ time. You can assume the arrays contain no duplicate elements.

Solution: The procedure `FindMedianSorted(A[1..n], B[1..n])` finds the median element of $A[1..n] \cup B[1..n]$ assuming the arrays are sorted and contain no duplicate elements.
FindMedianSorted(A[1 .. n], B[1 .. n]):
    if n = 1
        return min {A[1], B[1]}
    else if A[\lfloor n/2 \rfloor] < B[\lfloor n/2 \rfloor]
        return FindMedianSorted(A[\lfloor n/2 \rfloor + 1 .. n], B[1 .. \lceil n/2 \rceil])
    else
        return FindMedianSorted(A[1 .. \lceil n/2 \rceil], B[\lfloor n/2 \rfloor + 1 .. n])

If \( n = 1 \), then there are two elements total, and the algorithm is correct to return the smaller of them as the median. Otherwise, suppose \( A[\lfloor n/2 \rfloor] < B[\lfloor n/2 \rfloor] \). Consider any element \( A[i] \) where \( i \leq \lfloor n/2 \rfloor \). All \( \lfloor n/2 \rfloor \) elements from \( A[\lfloor n/2 \rfloor + 1 .. n] \) and all \( \lfloor n/2 \rfloor + 1 \) elements from \( B[\lfloor n/2 \rfloor .. n] \) are greater than \( A[i] \), so \( A[i] \) has rank at most \( n - 1 \) and it cannot be the median. Likewise, consider any element \( B[i] \) where \( i \geq \lceil n/2 \rceil + 1 \). Now, all \( \lfloor n/2 \rfloor \) elements of \( A[1 .. \lfloor n/2 \rfloor] \) and all \( \lfloor n/2 \rfloor \) elements of \( B[1 .. \lceil n/2 \rceil] \) are less than \( B[i] \), meaning it has rank strictly greater than \( n \) and cannot be the median. The median must belong to one of \( A[\lfloor n/2 \rfloor + 1 .. n] \) or \( B[1 .. \lceil n/2 \rceil] \). Further, because those subarrays are missing exactly \( \lfloor n/2 \rfloor \) elements smaller than the median of \( A[1 .. n] \cup B[1 .. n] \), it must have rank \( n - \lfloor n/2 \rfloor = \lceil n/2 \rceil \), making it the median of \( A[\lfloor n/2 \rfloor + 1 .. n] \cup B[1 .. \lceil n/2 \rceil] \). The recursive call \( \text{FindMedianSorted}(A[\lfloor n/2 \rfloor + 1 .. n], B[1 .. \lceil n/2 \rceil]) \) finds this element by induction on \( n \) and returns it. The final case of \( A[\lfloor n/2 \rfloor] > B[\lfloor n/2 \rfloor] \) is symmetric.

The algorithm running time follows the recurrence \( T(n) = T(n/2) + 1 \) which we just saw has a solution of \( O(\log n) \).

Rubric: 5 points total: 3 points for the algorithm; 1 point for the justification; 1 point for running time analysis. −0.5 points for missing the base case.
Describe backtracking recursive algorithms for the following variants of the text segmentation problem. Assume that you have a subroutine IsWord that takes an array of characters as input and returns True if and only if that string is a “word”. You do not need to analyze the time complexity of your algorithms for this problem.

(a) Given two arrays A[1..n] and B[1..n] of characters, decide whether A and B can be partitioned into words at the same indices.

Solution: The procedure BOTH_SPLITTABLE(A[1..n], B[1..n]) returns True if and only if strings A and B can be partitioned into words at the same indices.

```plaintext
BOTH_SPLITTABLE(A[1..n], B[1..n]):
if n = 0
    return True
for i ← 1 to n
    if IsWord(A[1..i]) and IsWord(B[1..i])
        if BOTH_SPLITTABLE(A[i+1..n], B[i+1..n])
            return True
return False
```

If n = 0, then we can split the two empty strings (into empty sequences of words). Suppose n > 0. Any pair of partitions at the same indices must each contain a first word ending at the same character i, and these first words must be followed by a pair of partitions of the remaining characters i + 1 through n at the same indices. The loop checks, for each pair of first words ending at the same index, if the remainder of the strings can be partitioned recursively. If no such first words exist, then there is no good pair of partitions, and we are correct to return False.

Rubric: 5 points total: 4 points for the algorithm and 1 point for the justification.

(b) Given an array A[1..n] of characters, compute the number of partitions of A into words.

Solution: The procedure COUNT_SPLITS(A[1..n]) returns the number of partitions of A into words.

```plaintext
COUNT_SPLITS(A[1..n]):
if n = 0
    return 1
count ← 0
for i ← 1 to n
    if IsWord(A[1..i])
        count ← count + COUNT_SPLITS(A[i+1..n])
return count
```
There is exactly one way to partition an empty string: into an empty sequence of words. So the algorithm is correct to return 1 when $n = 0$. Suppose $n > 0$. A partition of $A$ must begin with a single word $A[1..i]$ for some $i$. The number of partitions that begin with that word is then the number of ways to partition the remainder of the string $A[i+1..n]$. The algorithm loops over all choices of first word, adding up the recursively computed number of ways to partition the string given that first word.

\begin{center}
\textbf{Rubric:} 5 points total: 4 points for the algorithm and 1 point for the justification.
\end{center}