1. Assume that all sequences referred to in this question consist of distinct elements, from a totally ordered set. A sorting algorithm requires to output a sequence in increasing order. In this question, we are concerned with only comparison-based sorting algorithms.

A sequence $a_1, \ldots, a_n$ is monotonic (or one-monotonic) if it is a sorted sequence (ascending or descending). Examples: $1, 4, 5, 6, 8, 11, 19, 23$ (increasing) and $91, 37, 22, 7, 5, 3, 1$ (decreasing).

A sequence $a_1, \ldots, a_n$ is two-monotonic, if there exists an index $k \in \{2, \ldots, n-1\}$, such that $a_1, \ldots, a_k$ is increasing (respectively decreasing) and $a_k, \ldots, a_n$ is decreasing (respectively increasing). Examples: $1, 7, 15, 23, 40, 22, 6, 2$ (increasing, decreasing; index $\text{index} = 5$ is the turning point). $79, 36, 28, 21, 11, 7, 24, 30$ (decreasing, increasing; $\text{index} = 6$ is the turning point).

In more general terms, a sequence is $k$-monotonic, if there exist $k-1$ distinct indices $1 < i_1 < \cdots < i_{k-1} < n$, such that $a_1, \ldots, a_{i_1}$ is increasing (respectively decreasing), $a_{i_1}, \ldots, a_{i_2}$ is decreasing (respectively increasing), and so on alternatingly.

(a) Give a worst-case linear time ($O(n)$) algorithm to sort a monotonic sequence.

solution

If it is in ascending order do nothing. This decision can be made by comparing $A[1]$ with $A[2]$. If $A[1] < A[2]$ do nothing. This takes $\Theta(1)$ time. If $A[2] < A[1]$, then we need to reverse the elements. This can be done by having a for loop running from 1 to $\left\lfloor \frac{n}{2} \right\rfloor$. In each iteration of the for loop, exchange $A[i]$ with $A[n-i+1]$. This takes $\Theta(n)$ time in this latter case.

(b) Give a worst-case linear time algorithm to sort a 2-monotonic sequence.

solution

We are given that it is a 2-monotonic sequence. By comparing $A[1]$ with $A[2]$ we can determine if the initial portion is increasing or decreasing. If it is decreasing then the element $A[1]$ is the largest element in the first monotonic sequence and the element $A[n]$ is the largest element in the second monotonic sequence. In this case the two parts can be merged using the merge procedure of merge sort, where we take the pointers to the sequences to be merged to their respective maximum elements. We merge the two subarrays into a new array $B$ by placing the successive largest elements backward starting from $B[n]$.

In case the initial portion is increasing, then $A[1]$ and $A[n]$ represent the minimum elements of the left and right monotonic sequences respectively. In this case we again merge the two subarrays, but start storing the resultant minimum elements from $B[1]$ instead of $B[n]$. Note that we do not need to find the turning point explicitly. We just need to increment the pointer of the left subarray or decrement the pointer of the right subarray and always place the larger element into the output array (smaller in case of the increasing-decreasing case) until the array $B$ is full. This procedure takes $\Theta(n)$ time.

(c) What is the largest $k$ for which an $n$ element sequence is $k$-monotonic.

solution

The largest $k$ for which an $n$ element sequence is $k$-monotonic is $k = \lfloor \log_2 n \rfloor$. This is because the number of distinct indices $i_1, \ldots, i_{k-1}$ required to create a $k$-monotonic sequence is at most $k-1$, and $2^{k-1} \leq n$ implies $k-1 \leq \log_2 n$. Therefore, the largest $k$ is $k = \lfloor \log_2 n \rfloor$. 

[2 marks]
The monotonic parts are separated by indices representing a change in the trend of the sequence. These indices, called turning points can occur anywhere except the first or last position. Thus the number of turning points is at most $n-2$. The number of maximal monotonic sequences into which the given sequence can be broken is one more than the number of turning points. Thus the answer is $n-1$.

(d) Give as efficient an algorithm as you can for sorting a $k$-monotonic $n$-element sequence. Analyse its running time and express the time as a function of $n$ and $k$. The algorithm should be uniform and not vary for different values of $k$.

solution
We make a linear scan through the sequence comparing successive elements and separating the sequence into monotonic sequences at the turning points. The comparison of the first two terms yields the information of whether the first sequence is increasing or decreasing.

We then separate the original sequence into $k$ disjoint sequences. We pair of successive sequences in pairs and then merge each pair as in part $(b)$. now we are left with $\frac{k}{2}$ sequences each of which is sorted in increasing order. We again merge them by pairing them off. Thus it takes $\log k$ iterations to get a single sequence. The total cost of merging at each iteration is proportional to the total length of all the subarrays. This is thus $\theta(n)$. Thus the overall running time is $\theta(n \log k)$.

(e) Estimate the worst-case running time (as a function of only $n$), by taking the maximum value of the running time, ranging over all possible values of $k$, keeping $n$ fixed.

solution
As we have got a running time of $\theta(n \log k)$ in part $(d)$ and an upper bound of $n-1$ on $k$ in part $(c)$, we conclude that the worst case running time of such an algorithm is $\theta(n \log n)$.

(Hint:

i. If you get a running time better than $\theta(n \log n)$, the lower bound for any comparison based sorting algorithm in the worst-case, then there is something wrong in either your algorithm or your analysis.

ii. Try and use ideas in the merge-sort and merge procedures, for this question.)
2. The bubble sort algorithm sorts a sequence \( a_1, \ldots, a_n \) by repeatedly scanning through the sequence and swapping pairs of adjacent elements which are not in the correct order. The unit step is thus picking a pair of adjacent elements and comparing and either swapping or not swapping. The choice of which adjacent pairs to compare and the sequence of indices is carefully selected to ensure the algorithm performs correctly and as quickly as this basic operation allows. The bubble sort performs operations of the form \( a_i, a_{i+1} \leftrightarrow a_{i+1}, a_i \).

Consider the following proposed algorithm which is only allowed to perform operations of the form \( a_{i-1}, a_i, a_{i+1} \leftrightarrow a_{i+1}, a_i, a_{i-1} \). That is, the algorithm works by exchanging elements which are two places apart. Another way of viewing this is that substrings of length three are reversed. Can there be a sorting algorithm which uses only this kind of exchanges? If so, design such an algorithm. If not, then prove that no such algorithm can work successfully to sort an arbitrary input sequence.

[12 marks]

solution

No such algorithm can sort an arbitrary sequence correctly. In fact any such algorithm fails on an infinite set of inputs. Notice that by the definition of the basic operation, at any step the position of an element changes by either +2 or -2. Thus after any number of movements involving a particular element, that element has a displacement which is an even number. It follows that no element which starts out at an odd indexed position can ever reach an even indexed position and vice versa. Thus, the algorithm can never sort an input, where the source position of an element and its target position are of opposite parities (even or odd). Moreover, some such algorithm can always sort any input where each element is in a source position of the same parity as its destination position.

3. Solve the following recurrence relations:

\[ T(n) = 37T(n^{1/3}) + 4 \]

3 × 10 marks

(a) \( T(n) = 37T(n^{1/3}) + 4 \) solution

We recurse till we get a suitably small problem size and then apply a brute force solution. Notice that successively taking the cube root of a positive integer greater than 1, will never reach 1, in a finite number of iterations. Thus, it is necessary to take the boundary condition as at least 2. Taking it to be 2, we get, \( n^{1/3k} = 2 \) as the equation for the depth of the recursion tree. Here \( k \) is the depth. Solving yields, \( \frac{1}{3k} \log n = 1 \) (taking logarithm to the base 2 on both sides). Thus \( 3^k = \log n \) or \( k = \frac{\log \log n}{\log 3} \). Again, the base of the logarithm is 2. Note, that in this example the cost of each internal node is 4, which is \( \Theta(1) \). This is identical to the cost at each leaf, which is always taken as \( \Theta(1) \) reflecting the fact that, we recurse no further and solve the problem on a small sized problem in constant time using a brute force algorithm. Thus the cost of each internal node is the same as the cost of each leaf. Thus, we do not need to distinguish the leaf cost from the cost of the internal nodes. We get, \( T(n) = \sum_{i=0}^{k} 37^i \Theta(1) \), where \( k = \frac{\log \log n}{\log 3} \). Thus the running time is \( T(n) = \frac{37^{k+1} - 1}{36} \Theta(1) = \Theta(37^k) = \Theta((\log n)^{\log_3 37}) \).

(b) \( T(n) = T(k) + T(n-k) + 7 \) solution

Here, we treat \( T(k) \) as a constant and recurse no further. Thus, when the sub-problem size becomes at most \( k \), it is a leaf node and has cost \( \Theta(1) \). From the given recurrence, the cost of an internal node is \( 7 \), which is also \( \Theta(1) \). Thus, as in the previous example, here also the cost of an internal node is asymptotically the same as that of a leaf. Notice that every internal node in this recursion tree has exactly 2 children. Thus, it is a full binary tree. As demonstrated/proved in some of the lectures, every full binary tree has one more leaf than its number of internal nodes. Thus the set of leaves has approximately the same size as the set of internal nodes. Taking \( T(k) = \Theta(1) \), the total cost is proportional to the total number of nodes. We start from an initial problem of size \( n \). At each level of recursion, we hit one leaf and a subproblem in an internal node of size \( k \) less. Thus the total number of leaves is approximately \( \frac{n}{k} \). The total number of nodes is thus, approximately \( 2 \frac{n}{k} \). Each node has cost \( \Theta(1) \). Thus the total cost, \( T(n) = \Theta(\frac{n}{k}) = \Theta(n) \), since \( k \) is a constant.

(c) \( T(n) = 6T(5n/8) + n^3 \) solution

This can be solved by the Master Method. Here, we have \( a = 6, b = \frac{8}{5} \) and \( f(n) = n^3 \). We see that \( (\frac{8}{5})^3 = \frac{512}{125} < 6 \). Thus, \( \log_\frac{8}{5} 6 > 3 \). Thus \( f(n) = n^3 = O(n^{\log_\frac{8}{5} 6 - \epsilon}) \), for some \( \epsilon > 0 \). This falls into case 1 of the Master Method. Thus the solution to the recurrence is therefore, \( \Theta(n^{\log_\frac{8}{5} 6}) \).

You may assume, the function \( T(n) \) is \( \Theta(1) \) for sufficiently small values of \( n \).
4. Consider a generalisation of a binary heap, called a \( k \)-ary heap, wherein, the heap property (the key of a node is greater than the key of its children) is maintained, but each internal node has \( k \) children instead of 2 children. Like in a binary heap, a \( k \)-ary heap is a complete \( k \)-ary tree except, possibly, at the last level, where the nodes are filled from left-to-right.

(a) Describe how a \( k \)-ary heap can be stored in a single one-dimensional array. How are the indices of the \( i^{th} \) child of a node calculated, for \( i \in \{1, \ldots, k\} \). How is the parent’s index calculated.

**solution** The heap is stored in the array by placing the elements in order level by level and within each level starting from the leftmost node till the rightmost node. The only difference from the binary heap is that each internal node, except the last such one has \( k \) children instead of 2.

Let us assume that the root is stored at array position \( A[1] \). Any node can be uniquely described by its level number (root has level number 0) and its number from the left within that level. Thus a node may be at level \( l \) and may be the \( j^{th} \) node from the left, in that level. The index of that element in the array can then be calculated as \( j + \text{total number of nodes up to level } l - 1 \). This is equal to \( j + \sum_{i=0}^{l-1} k^i = j + \frac{k^{l+1}-1}{k-1} \). The index of its \( i^{th} \) child is \( i \). The total number of nodes up to and including level \( l + (j - 1)k + i = \sum_{i=0}^{l} k^i + (j - 1)k + i = \frac{k^{l+1}-1}{k-1} + jk - k + i = k \left( j + \frac{k^{l+1}-1}{k-1} \right) + (1 - k) + i = k \times \text{index of node} + (1 - k + i).

It follows immediately, that the parent of a node of index \( x \) is \( \lceil \frac{x - 1}{k} \rceil \).

(b) Give a description of how to build a heap of this type from an arbitrary array of \( n \) elements. Analyse the running time of your procedure.

**solution**

All the leaf nodes are roots of subtrees containing only one element. Thus they are trivially heaps. We run a modified heapify procedure working backwards from the last internal node sequentially to the root. At every stage, we assume the subtrees rooted at the children of a node are heaps, and the heap property may be violated at that node itself. We find the node with maximum key value among the children of a node. We then compare that value with the value of the key at the node itself. If the value of the maximum child is greater, we exchange the values of the node and the child. In this case we run heapify at that child node. This happens until we either reach a leaf node or the value of the parent node is greater than that of all its children. Heapify at a node involves \( k \) comparisons. To calculate the total cost of build heap, we will calculate the total number of recursive calls to heapify, at a level, rather than summing over the runtime of heapify originating at some node. The total number of times heapify may be called recursively at nodes of height 1, is at most the number of non-leaf nodes. This is given by \( \lceil \frac{n-1}{k} \rceil \). The total number of times heapify may be called recursively at nodes of height 2 is likewise given by \( \lceil \frac{n-1}{k} \rceil /k = \lceil \frac{n-1}{k^2} \rceil \), and so on. Thus the total cost is \( O \left( \sum_{i=1}^{\infty} \lceil \frac{n-1}{k^i} \rceil \right) = O(n) \), since \( k \) is a constant and it is a decreasing infinite geometric series. A less intricate calculation results in the looser upper-bound \( O \left( \frac{n}{k} \log_k n \right) = O(n \log n) \).

(c) Describe a heapsort algorithm on the basis of this heap. Analyse its running time on \( n \) elements.

**solution**

Build a heap by the procedure described in part (b). Starting from position \( n \) down to 2, exchange that element with the root, thus placing the root value (maximum element) at the end of the heap. Reduce the size of the heap by one. Apply heapify at the root, with new key value. We get a new heap on one element less. This procedure results in the entire array getting sorted. The cost is \( O(\log_k n) \) for each heapify and this is done \( \theta(n) \) times. The build heap costs \( \theta(n) \) as we have seen. Thus the cost is \( \theta(n \log_k n) = \theta(n \log n) \). The lower bound is because, we know every comparison based sorting algorithm has running-time \( \omega(n \log n) \).

5. An absolute majority element in a multiset of \( n \) elements is any element which occurs at least \( \lceil \frac{2n}{3} \rceil \) times. Obviously, any set can have at most one absolute majority element. Give a worst case \( \theta(n) \) time algorithm to find an absolute majority element of an \( n \) element multiset given as input, if one exists, and reporting no otherwise.

**solution**

We know that every absolute majority element is also a majority element. Thus it suffices to search for a majority element and if one exists, to check if it occurs at least as frequently as the qualification mark for being an absolute majority element.
We scan pairs of adjacent elements in the order \((1, 2), (2, 3), \ldots, (n - 1, n)\). Every time we encounter a pair of distinct elements, we throw away both from the multiset, and continue scanning to the right of the second element of that pair. It is easy to see that if either of them is a majority element of the original set then it continues to be a majority element of the reduced set. We continue scanning on the reduced set with a comparison between the elements just to the left and the right of the eliminated pair. The left element has an indicator to reflect the fact that its successor is obtained by a jump. In the end we are either left with one element or two elements or a large block of identical elements. In all these cases, the remaining elements are the only candidates for the absolute majority element. We need to scan the original array for the frequency of their occurrence and declare them as absolute majority element if that frequency is at least \(\lceil \frac{2n}{3} \rceil\) and report the nonexistence of an absolute majority element otherwise.