CS5592 Problem Suggestions

First Some background. Questions follow below

Suppose you have an array $MyA[1 \ldots n]$ with n unique integers in a random order. A *peak* in this array is an element that is larger than its immediate neighbors. Thus, the element at location i is a peak if

$$A[i] \text{ is a peak} \Leftrightarrow \begin{cases} A[1] > A[2] & i = 1 \\ \text{both } A[i-1] < A[i] \text{ and } A[i+1] < A[i] & 1 < i < n \\ A[n] > A[n-1] & i = n \end{cases} \quad (1)$$

For example, the array $\begin{bmatrix} 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 & 12 & 13 & 14 & 15 & 16 & 17 & 18 \\ 5 & 9 & 8 & 11 & 14 & 13 & 23 & 36 & 29 & 14 & 53 & 63 & 34 & 26 & 27 & 15 & 12 & 17 \end{bmatrix}$ has peaks at locations 2, 5, 8, 12, 15, and 18. Note, that the number of peaks is between 1 and $n/2$, and that the maximum is also a peak, but not vice versa. The problem of *find-a-peak* in an array, is the identification of an index where a peak occurs, without disturbing the structure of the array. A straightforward algorithm is going through the array linearly.

algorithm `FPSlow(array` MyA`)`
$i \leftarrow 1$
while ($MyA[i] < MyA[i+1]$ **and** $i \leq MyA$.SIZE) **do** $i \leftarrow i + 1$ **end-while**
return(min(i,n))
end algorithm

The worst-case time complexity is $T_A^W(n) = n - 1$, thus linear. However, another algorithm has a logarithmic performance. This algorithm is correct, and made possible by the observation that there may be multiple peaks inside an array, and *any* peak can be returned.

algorithm `FPFast(array` MyA`)`
if $n == 2$
  then use 1 comparisons to find location of the peak
  else Let m be the middle element
    // Use 2 comparisons to decide between three cases:
    whenever $A[m-1] < A[m]$ and $A[m+1] < A[m]$ **then return** (m), a peak has been found
    whenever $A[m-1] > A[m]$ **then return** (`FPFast(left half of` MyA`)`), a peak is at left half
    whenever $A[m+1] > A[m]$ **then return** (`FPFast(right half of` MyA`)`), a peak is at right half
end algorithm

The worst case time complexity is given by $T_{FP}^W(n) = \begin{cases} 1 & n = 2 \\ 2 + T^W(\frac{n}{2}) & n = 2^L, \text{ for some } L > 1 \end{cases}$ with solution $T^W(n) = 2 \lg n \ - 1$.

The notion of a *valley* is similarly introduced, the number of valleys is similarly between 1 and $n/2$, and the location of a valley is similarly determined with algorithms called `FVSlow` and `FVFast` . Let us now consider the problem is determining the location of both a peak and a valley simultaneously. We will consider several algorithms for this, and call them all `P&V`.

The questions I would like you to think about are on the next pages.

---

1. This is a variation and you are asked to argue why it is correct (or give a counter-example), find the explicit expression for the best and worst cases, and describe the particular array configuration that give rise to this best/worst case time complexity.

algorithm `P&V.v1(array` MyA`)`
if $n == 2$
then use 1 comparisons to find location of the peak and of the valley.
else return (`FP`(all of MyA), `FV`(all of MyA))
end-if
end algorithm

---

2. This is a variation and you are asked to argue why it is correct (or give a counter-example), find the explicit expression for the best and worst cases, and describe the particular array configuration that give rise to this best/worst case time complexity.

algorithm `P&V.v2(array` MyA`)`
if $n == 2$
then use 1 comparisons to find location of the peak and of the valley.
else $p \leftarrow$ `FP`(all of MyA)
return(p,`FV`(the smaller of either left or right of p in MyA))
end-if
end algorithm

---

3. This is a variation and you are asked to argue why it is correct (or give a counter-example), find the explicit expression for the best and worst cases, and describe the particular array configuration that give rise to this best/worst case time complexity.

algorithm `P&V.v3(array` MyA`)`
if $n == 2$
then use 1 comparisons to find location of the peak and of the valley.
elseif $A[1] > A[2]$
then return (1, `FV`(elements 2 through n of MyA)), // a peak is at location 1
else return (`FP`(elements 2 through n of MyA),1), // a valley is at location 1
end-if
end-if
end algorithm

---

4. This is a variation and you are asked to argue why it is correct (or give a counter-example), to find the explicit expression for the best and worst cases, and describe the particular array configuration that give rise to this best/worst case time complexity.

algorithm `P&V.v4(array` MyA`)`
if $n == 2$
then use 1 comparisons to find location of the peak and of the valley.
else Let m be the middle element
Use 2 comparisons to decide between four cases:
when $A[m-1] < A[m]$ and $A[m+1] < A[m]$ **then return** (m, `FV`(any half of MyA)), // a peak is at m
when $A[m-1] > A[m]$ and $A[m+1] > A[m]$ **then return** (`FP`(any half of MyA), m), // a valley is at m
when $A[m-1] < A[m] < A[m+1]$ **then return** (`FP`(right half of MyA), `FV`(left half of MyA)), p/v at r/l
when $A[m-1] > A[m] > A[m+1]$ **then return** (`FP`(left half of MyA), `FV`(right half of MyA)), p/v at l/r
end algorithm

---

5. Could you find yet another version with similar time complexity?

---

6. From the hand out on recurrence relations, do
questions 7 through 15 on page 29/30,
exercises 2, 4, 6, 8, 10, 12, and 13 on pages 30/31, and
page 36: numbers 3, 6, and 9.

7. Review the algorithm (MIN2) that determines both the first and the second smallest element from an array in time $\sim n + \lg n$, see below. Analyze (for worst case only) an algorithms that determines the top 3 elements, by first finding the MIN using the tournament method, then finding the *min* of the losers to winner by linearly going through the "losers to the winner", and then finally finding the third MIN by linearly going through "distance two losers to the winner" AND the "losers-to-the-runner-up". You should assume that n is a power of 2.

---

8. Review the algorithm (MIN2) that determines both the first and the second smallest element from an array in time $\sim n + \lg n$, see below. Analyze (for worst case only) an algorithms that determines the top 3 elements, by first finding the MIN using the tournament method, and then using the tournament method again to find the runner-up from among the "distance two losers to the winner", after which the third place can be found using a linear scan from the remaining "distance two losers to the winner". You should assume that n is a power of 2.

---

9. Review the algorithm (MIN2) that determines both the first and the second smallest element from an array in time $T(n) \sim n + \lg n$, see below. Also, review the algorithm (MINMAX) that determines both the min- *and* the max in a single algorithm, and which has time complexity $T(n) \sim \frac{3}{2}n$. Now design a new algorithm that determines both the top 2 elements, as well as the bottom 2 elements in $T(n) \sim \frac{3}{2}n + 2\lg n$. You must argue correctness of your approach, which you should call MIN2MAX2. You should assume that n is a power of 2.

Towards a good Algorithm to Select the rank k element

Let x_k be the rank-k element of an array (or set) with n elements. That is, it is such that there $k-1$ elements that are smaller than x_k (and thus there are $n-k$ elements that are larger). Later in the semester, we will present an algorithm whose worst time complexity is linear, although the asymptotic region is "far away". Meanwhile, please find the worst case time complexities for the following algorithmic ideas. Also, assume that we are looking for the median, that is, assume that $k = \frac{1}{2}n$.

---

10. Sort the array and return the element with the middle index.
11. Repeatedly find (& remove) the MIN-element.
12. Repeatedly find (& remove) the two MIN2-elements.
13. Repeatedly find (& remove) the three MIN3-elements.
14. Repeatedly find (& remove) the two MINMAX-elements.
15. Repeatedly find (& remove) the four MIN2MAX2-elements.
16. Create HEAP with all n elements, and remove k elements.
17. Create HEAP with only k elements, now HEAP.INSERT the remaining elements. The median is now at the top of the heap.
18. Use QUICKSELECT where the pivot is the first element.
19. Use QUICKSELECT where the pivot is the median of three elements.
20. Use QUICKSELECT where the pivot is the median of five elements.

---

21. The Quicksort algorithm is the industry standard even though there are quite a few variations. The variation used at your place of employment uses a pivot selection of "*median-of-three-random-elements*". It has a excellent performance of both best - and average cases of $T^B(n) = \Theta(n\ \log n)$ and $T^A(n) = \Theta(n\ \log n)$, and a worst case of $T^W(n) = \Theta(n^2)$.
Part 1. Find the coefficient c so that $T^W(n) \sim c\, n^2$
Part 2. This worst case apparently seems to occur more frequently than the management had anticipated and the management desperately wants to improve the worst case performance and get an edge on the competition. You have been selected to implement the variation where a pivot selection of "*median-of-seven-random-elements*". The algorithm as designed by your manager is now:

algorithm QUICKSORT.vMED7(MyA)
if $MyA.size < 7$
  then Use BubbleSort to sort this small array
  else-do $pivot \leftarrow$ MED7
    {Partition($MyA, pivot$)
    QUICKSORT.vMED7($MyA.left$)
    QUICKSORT.vMED7($MyA.right$)
    end
  end if
end algorithmQUICKSORT.vMED7
Given that $T^W_{\text{MED5}}(n) = 6$ and that $T^W_{\text{MED7}}(n) = 10$, what is the dominant term of $T^W_{\text{QUICKSORT.vMED7}}(n)$

Part 3. Although you had a full day to implement the MED7 algorithm, you can not seem to get it right, and you also use BubbleSort to sort these seven elements, and return the middle as pivot. What is the dominant term of your implementation?

Determine first and second place winners

An algorithm to determine both Min and the second min, which we called the "**algorithm** `MIN2`" algorithm, was discussed earlier in class, and repeated here for convenience. It uses a **Complete Binary Tree**, which is easily implemented as an Array: The parent of a node stored at array-location j is located at $j/2$ (integer division, remainder discarded), while the left- and right child of a node at array location j are at array locations $2j$ and $2j+1$. (A heap uses a similar structure: a complete binary tree). Let us assume that the size n is a power of 2, for ease of discussion, then store the indices 1 through n in the last half of the array. Now compare the values of neighboring indices (i.e. children) and place the index of the *winner* in the parent index. Continue in this fashion, till the winner (i.e. minimal value in this case) has been determined. The second place winner must have been compared with (and lost out to) the ultimate winner (why?), and this second place winner (runner up) can be found among the losers of the ultimate winner. These second place candidates are at "sibling" index locations for the ultimate winner as the winner progressed in the tournament. 'These second place candidates are thus generated again.

algorithm `MIN2`(MyA)
(* Precondition: The array MyA has n elements, with 0 the index of the first element
Define local array $MyIndex[1..2n-1]$ %% the variation of min-heap
Helper process: `MySibling` ::= **process** (k) **if** k is even **then return**$(k+1)$ **else return**$(k-1)$ **end-process**

for i **from** 0 **to** $n-1$ **do** $MyL[n+i] \leftarrow i$ **endfor**

for j **from** $n-1$ **downto** 0 **do**
  if $MyA[\ MyL[2j]\ ] < MyA[\ MyL[2j+1]\ ]$
  then $MyIndex[j] \leftarrow MyIndex[2j]$
  else $MyIndex[j] \leftarrow MyIndex[2j+1]$
  end if
endfor

//The winning element is at array index $MyIndex[0]$ and has value $MyA[\ MyIndex[0]\ ]$
$WinIdx \leftarrow MyIndex[0]$
$WinVal \leftarrow MyA[\ MyIndex[0]\ ]$

//The winning element started at index $n + MyIndex[0]$
//The first comparison was with $RunnerUpVal$, at index $RunnerUpIdx$
$w \leftarrow n + MyIndex[0]$ // The location of the "first" comparison.
$RunUpIdx \leftarrow$ `MySibling`(w) // The 'other" element of the "first" comparison.
$RunnerUpVal \leftarrow MyA[RunnerUpIdx\ ]$
while $w > 3$ **do** //as long as there is another element that was compared with (and "lost" to) $WinIdx$
  $w \leftarrow w/2$
  $s \leftarrow$ `MySibling`(w)
  if $MyA[\ MyIndex[s]\ ] < RunnerUpVal$ **then**
    $RunnerUpIdx \leftarrow MyIndex[s]\ ]$
    $RunnerUpVal \leftarrow MyA[\ RunnerUpIdx\ ]$
  end if
end while

return$[WinVal, RunnerUpVal]$

end algorithm `MIN2`

For an analysis of this approach, observe that the $WinVal$ is determined after $n-1$ comparisons. After that there are $(\lg n)-1$ comparisons needed to find the $RunUpVal$, for a total of $T_{\texttt{MIN2}}(n) = n - 2 + \lg n \sim n + \lg n$