Peak Finding

Peek Finding 1D Array

Source: * easy + takes little time * MIT + 1hr, ppt/slide

Let, $ar[] = \{a_0, a_1, a_2, ..., a_{n-1}, a_n\}$, then a_0 is a peak if $a_1 \leq a_0$ or a_n is a peak if $a_{n-1} \leq a_n$ or any arbitary element a_m from the elements other than the the first or last one if $a_{m+1} \leq a_m$ and $a_{m-1} \leq a_m$, in other words if an element is greater than both of it's adjacent elements. #### Warning!!! Need to check Greather than or 'equal to'?

Following corner cases give better idea about the problem: 1. If input array is sorted in strictly increasing order, the last element is always a peak element. For example, 50 is peak element in {10, 20, 30, 40, 50}. 2. If input array is sorted in strictly decreasing order, the first element is always a peak element. 100 is the peak element in {100, 80, 60, 50, 20}. 3. If all elements of input array are same, every element is a peak element.

pseudo code: Look at the comments to get the pseudo code.

```
// A C++ program to find a peak element element using divide and conquer
// A binary search based function that returns index of a peak element
int findPeakUtil(int arr[], int low, int high, int n)
{
    // Find index of middle element
    int mid = low + (high - low)/2;  /* (low + high)/2 */

    // Compare middle element with its neighbours (if neighbours exist)
    if ((mid == 0 || arr[mid-1] <= arr[mid]) &&
            (mid == n-1 || arr[mid+1] <= arr[mid]))
        return mid;

    // If middle element is not peak and its left neighbour is greater  than it, then left half must have
    else if (mid > 0 && arr[mid-1] > arr[mid])
        return findPeakUtil(arr, low, (mid -1), n);

    // If middle element is not peak and its right neighbour is greater than it, then right half must have
    else return findPeakUtil(arr, (mid + 1), high, n);
}
int findPeak(int arr[], int n)
{
    return findPeakUtil(arr, 0, n-1, n);
}
int main()
{
    int arr[] = {1, 3, 20, 4, 1, 0};
    int n = sizeof(arr)/sizeof(arr[0]);
    printf("Index of a peak point is %d", findPeak(arr, n));
}
```

Time Complexity: $O(log_2(n))$ Space Complexity: $O(1)$ Recurrence: $T(n) = T(\frac{n}{2}) + O(1)$

Proof:

If `ar[currentElement+1] >= ar[currentElement]` then the solution exists in the right part. Why? Because `ar[currentElemenet+1]` already has neighbour to its left which is smaller than smaller or equal. Then there are two cases. Either the `ar[currentElemenet+1]` is the last element, in that case it's our answer to the problem, or there are still some elements remaining after this point. At a point either there will be an element which will be less than it's previous element, then the previous element will be the answer. Or the rest of the sequence is a increasing sequence and that means that the last element is our answer to the problem. Refer to the sources mentioned above to get a better idea.

Peek Finding 2D Matrix

Source: * MIT Lecture Slide * GeeksForGeeks