CSCI 3104-Spring 2016: Assignment #1.

Assigned date: Monday 1/18/2016,
Due date: Tuesday, 1/26/2015, before class
Maximum Points: 50 points (includes 5 points for legibility).

Note: This assignment *must be turned in on paper, before class.* Please do not email: it is very hard for us to keep track of email submissions. Further instructions are on the moodle page.

P1 (Ternary Search, 30 points) We wish to search large array of n sorted integers for the number k. Instead of implementing *binary search* routine, we formulate an advanced search routine called ternary search.

Here is the overall idea of ternary search to check if a sorted list a of size n has the number k in it: (a) Ternary search divides the list a into three roughly equal parts of size $\frac{n}{3}$ (rounded up/down). (b) It then looks at the two elements $a[n/3]$ and $a[2*n/3]$. (c) If $k < a[n/3]$ it searches in the sublist from 0 to $n/3$, else if $a[n/3] \leq k < a[2*n/3]$ search in the sublist from 0 to $2*n/3$; otherwise search the the list from $2*n/3$ to n.

A recursive Python implementation is given below:

```python
def ternarySearch(a,k):
    n = len(a)
    if (n == 0):
        return False
    if (n <= 2):
       for i in range(0,n):
           if (a[i] == k):
               return True
       return False
    t1 = n/3
    t2 = 2*n/3
    if (a[0] <= k and k < a[t1]):
        return ternarySearch(a[0:t1],k)
    elif (a[t1] <= k and k < a[t2]):
        return ternarySearch(a[t1:t2],k)
    elif (a[t2] <= k and k <= a[n-1]):
        return ternarySearch(a[t2:n],k)
    else:
        return False
```

(A, 5 points) Demonstrate the working of the program on the function call by listing the recursive calls (function called and arguments to that call).

`ternarySearch( [1,3,5,10,12,15,32,91, 125,132], 18).`

Solution.

1. `ternarySearch( [1,3,5,10,12,15,32,91, 125,132], 18)`: We find that $10 \leq 18 \leq 32$.

2. `ternarySearch( [10,12,15], 18 )`: returns `False`.

▲

(B, 15 points) Prove the theorem that `ternarySearch` is correct.

Theorem: The procedure `ternarySearch(a,k)` returns `True` if and only if k is contained in a, and returns `False` otherwise.

Use strong induction on the size of the array a that is input to the `ternarySearch`. *Clearly write down the base case and induction hypothesis.*
Your proof does not need to exceed more than 3/4ths of a page.

Solution.

Proof: Proof is by strong induction on n the size of the list a. **Base Case:** For a list of size $n = 0$, we observe that the procedure correctly returns `False` since k cannot be in the empty list.

Induction Hypothesis: If for all lists l of size $0 \leq i \leq n$, the procedure `ternarySearch(l,k)` is correct, the procedure is also correct on any list a of size $n + 1$.

Proof. Let a be any given list of size $n + 1$ and k be any integer. We split on two cases:
Case-1: $n \leq 2$ where the procedure directly uses a non-recursive scan of the list. We notice that the scan individually searches all elements of the list and returns `True` if and only if k is present in the list a.
Case-2: $n \geq 3$. In this case, we compare k with $a[n/3]$ and $a[2 * n/3]$. First, because a is a sorted list, we split on three cases:

1. $a[0] \leq k < a[n/3]$: In this case, we note that k belongs to the list a if and only if it belongs to $a[0 : n/3]$. Assuming by induction hypothesis that the recursive call on an array of size at most $\lfloor \frac{n+1}{3} \rfloor$ `ternarySearch(a[0:n/3],k)` is correct, the call `ternarySearch(a,k)` also returns the same answer, which is the correct answer.

2. $a[n/3] \leq k < a[2 * n/3]$: In this case, we note that k belongs to the list a if and only if it belongs to $a[n/3 : 2 * n/3]$. Once again, the recursive call is over an array of size at most $\lfloor \frac{n+1}{3} \rfloor$. Assuming by the induction hypothesis, that the call returns the correct answer, we find that the overall call to `ternarySearch(a,k)` also returns the same answer, which is the correct answer.

3. $a[n] \geq k \geq a[2 * n/3]$: Once again, we note that k belongs to the list a iff it belongs to $a[2*n/3 : n]$. Once again, the recursive call over an array of size at most $\lfloor \frac{n+1}{3} \rfloor$ returns the correct result by the induction hypothesis. We find that therefore `ternarySearch(a,k)` also returns the same answer as the recursive call, which is the correct answer.

□

▲

(C, 10 points) For an array of size n, let $T(n)$ denote the number of recursive calls made by `ternarySearch` in the worst case. Derive a recurrence for $T(n)$ and solve it to find a tight Θ-bound on $T(n)$.
If you are on the right track, the answer should have 5 lines or less.

Solution. Let $T(n)$ denote the number of recursive calls of ternary search program. Looking at the program, we derive the following recurrence for the worst-case:

$$T(n) = \begin{cases} 0 & \text{If } n \leq 2 \\ 1 + T(\lceil \frac{n}{3} \rceil) & \text{Otherwise} \end{cases}$$

Expanding this recurrence, we obtain:

$$T(n) = 1 + T(\lceil \frac{n}{3} \rceil) = 2 + T(\lceil \frac{n}{3^2} \rceil) = \cdots = k + T(\lceil \frac{n}{3^k} \rceil)$$

For $k = \lceil \log_3(n) \rceil$, we obtain:

$$T(n) = k + T(\underbrace{\lceil \frac{n}{3^k} \rceil}_{\leq 2})$$

Thus, we obtain $T(n) = \lceil \log_3(n) \rceil$. In other words, $T(n) = \Theta(\log_3(n))$.

▲

P2, 20 points An integer array a of size N is *all but k sorted* if k elements are *out of position*.

> An element $a[j]$ is *out of position* if and only if there is an index $l < j$ such that $a[l] > a[j]$.

For instance,

$$[1, 3, 5, 2, 7, 10, 8]$$

is all but 2 sorted with the two elements that are out of position shown.

Another example,

$$[1, 2, 5, 10, 11, 7, 8, 13, 12]$$

is all but 3 sorted with the three elements out of place shown.

(A, 5 points) Write down pseudocode (or Python code) to check if an array is all but k sorted. You are given as inputs the array a and the number k. What is the running time of your algorithm in the worst case?

(B, 10 points) Prove that the running time of *insertion sort* algorithm on a given list a of size N that is all but k sorted is bounded by $O(N \times k)$.

Solution. Consider a run of insertion sort on all but k sorted list a. At the j^{th} step of the algorithm, we always have the following situation:

The sublist $a[0]$ *to* $a[j-1]$ *is sorted and the element* $a[j]$ *is to be inserted into this sublist.*

Therefore, two cases arise:

1. $a[j]$ is in its correct position in the sorted order. And therefore, the insertion takes $O(1)$ time.

2. $a[j]$ is out of position and therefore, the insertion takes $O(j)$ time.

However, since exactly k elements are out of position, the case 2 above can occur exactly k times. Let $j_1, \ldots, j_k$ be the out of position elements in the original list a. The running time is therefore $O(N + \sum j_i)$. This is $O(k \times N)$. ▲

(C, 5 points) Argue why the recursive version of `mergeSort` presented in class can still take $O(n \log n)$ time even if it's input is already a fully sorted array? How can this be fixed?

Solution. The reason is that the recursive `mergeSort` has two issues: (a) If we call `merge`(a, b) on two sorted lists a, b, and the first element of b is already larger than the last element of a, merge can simply concatenate the two lists and return. (b) Each merge physically copies its elements into a new list and therefore, it takes time that is linear in the size of the input lists.

One solution is to first run a scan through the list and find out how many elements are out of position. If this number is less than a constant k, we use insertion sort. Failing that, we may use `mergeSort` routine.

▲