Jason Caplan

Design and Analysis of Algorithms

Professor Leff

HW6

1. Iterate through the array of viruses, and for each one, scan the entire rest of the array and count how many other viruses are equivalent to this one, returning the current index if the count exceeds $\frac{n}{2}$. (Really we can stop after the outer loop gets past the $(\frac{n}{2})^{\text{th}}$ element, but this does not effect the Big-O). The performance will be $O(n^2)$, since for each of the n elements we check $n-1$ other elements.

2. We will use a standard divide-and-conquer approach wherein we split the problem into two parts, recursively solve these sub-problems, and use a relatively cheap (in this case $O(n)$) step to combine the results to arrive at the result of the larger problem. (In other words, a merge-sort flavor). In this case, we split the array into a left half and a right half, find the "most prevalent" of each sub-array internally (if it exists), and then use the following combination logic to determine the most prevalent of the combined array:

    a. The return of each recursive call contains a wrapper object that holds a canonical representative of the "most prevalent" equivalence class, its index, and the integer number of members of that class that were found in the array being checked. If no "most prevalent" is found, null is returned
    b. If the "most prevalent" equivalence class in the 2 sub-arrays are equivalent to each other, this equivalence class is trivially the "most prevalent" in the combined array
    c. If a given equivalence class of viruses was not returned as the "most prevalent" in either of the sub-arrays, it surely cannot be the "most prevalent" of the combined array, so the only candidates to be "most prevalent" in the combined array are the equivalence classes represented by the returned result from each subarray.
        i. We can easily check if the left-sub-array's "most prevalent" is also the combined array's "most prevalent" by adding the number of members found in the left-subarray to the result of counting the members in the right-sub-array (an $O(n)$ operation). The same can be done to check the result of the right-sub-array.
        ii. If we find a new "most prevalent", we return a wrapper object with an updated count of the size of the "most prevalent" equivalence class

3. (see below)

Pseudo-code

```
1       # makes top-level call to recursive method
2       mostPrevalent(virus_database):
3           wrapper_object = find(virus_database[0→end])
4           return wrapper_object.index if not null, otherwise NOT_FOUND
5
6       # recursive method, makes 2 recursive calls on problems of half the size
7       find(interval):
8           if (interval.size < small constant c):
9               # hand off to O(1) brute-force
10              return solve(interval)
11
12          # recursively find "most prevalent" for each sub-array
13          left_result = find(left half of interval)
14          right_result = find(right half of interval)
15
16          # combine step
17          if (left_result not null):
18              # check if left_result.Virus is "most prevalent"
19              # in the combined array
20              right_count = 0
21              for each Virus in right half that is "equal to"
22                  left_result.Virus, increment right_count
23              total_count = left_result.count + right_count
24              if (total_count > interval.size / 2):
25                  return left_result with count = total_count
26          if (right_result not null):
27              # check if right_result.Virus is "most prevalent"
28              # in the combined array
29              left_count = 0
30              for each Virus in left half that is "equal to"
31                  right_result.Virus, increment left_count
32              total_count = right_result.count + left_count
33              if (total_count > interval.size / 2):
34                  return right_result with count = total_count
35
36          return null   # neither candidate was most prevalent
37
```

Base Case → O(1)

2 recursive calls, halve the size of the data for each, so 2T(n/2)

Iterative Work → O(n)

```
38      # O(1) brute-force method to solve for "most prevalent"  of interval
39      # of a predetermined constant length c (e.g. c=6)
40      solve(small_interval):
41          for (Virus_a in small_interval):
42              count = 0, index = index of Virus_a
43              for (each other Virus_b in small_interval):
44                  if (Virus_a is "equal to" Virus_b) increment count
45              if (count > small_interval_length / 2):
46                  return new wrapper(Virus_a, count, index)
47          return null  # no "most prevalent" Virus
```

Takes $\boldsymbol{O(c^2)}$, so $\boldsymbol{O(1)}$

Recurrence Relation

Each call to **find** consists of an $O(1)$ call to **solve** in the base case (since the brute force for a constant size c is $O(c^2) = O(1)$), otherwise two recursive calls with intervals half as large, and an $O(n)$ iterative combination step. This yields the following recurrence for $Q(n)$:

$$Q(n) = \begin{cases} O(1) & n \leq c \\ 2Q\left(\frac{n}{2}\right) + O(n) & n > c \end{cases}$$

Performance

Applying the Master Theorem to the aforementioned recurrence relation,

$$a = 2, \quad b = 2, \quad d = 1$$

$$2 = 2^1, so$$

$$Q(n) = \mathbf{n} \times \mathbf{log}(\boldsymbol{n}) \ \blacksquare$$

Correctness

NOTE: For this proof, assume all division using $\frac{a}{b}$ notation is Java integer division, and division using a/b notation is standard real-number division.

Also, we quickly prove the following 2 short lemma's we'll use later:

Lemma 1a: for any $k \in Z^+$, $\frac{2k+1}{2}$ will always resolve to k

Direct Proof:

$$\frac{2k+1}{2} = \lfloor (2k+1)/2 \rfloor = \lfloor 2k/2 \ + 1/2 \rfloor = k \ \blacksquare$$

Lemma 1b: for any $k \in Z^+$, $\frac{k}{2} + \frac{k+1}{2}$ will always resolve to k

Proof By Cases:

If k is even, we can express k as 2c, and substitute:

$\frac{2c}{2} + \frac{2c+1}{2} = c + c\ (by\ Lemma\ 1a) = 2c = k$

If k is odd, we can express k as 2c+1, and substitute:

$\frac{2c+1}{2} + \frac{2c+2}{2} = \frac{2c+1}{2} + \frac{2(c+1)}{2} = c\ (by\ Lemma\ 1a) + (c+1)\ = 2c + 1 = k$ ■

- First we show that the brute-force algorithm used for small sub-problems is correct
  - Since this method checks each Virus to see if its equivalence class's size is more than half of the interval size, it is definitionally solving the stated problem for the given interval, and returns a wrapper object that contains a canonical Virus from this equivalence class, its index, and the size of the equivalence class to be used further up the recursion tree.
- Next we show that, in all cases, combining 2 sub-solutions will yield a correct combined solution, which means that the entire problem is solved once all the recursions bubble up.
- We already expressed in "Part 2" that there are a maximum of 2 candidates for "most prevalent," the returned result from the left, and the returned result from the right. We now formalize this:
  - For an equivalence class to be the "most prevalent" in an array of size n, it must contain <u>at least</u> $\frac{n}{2} + 1$ Viruses.
  - It follows that if a given equivalence class EC was NOT returned by a recursive call to a sub-array of size m, it can contain <u>at most</u> $\frac{m}{2}$ Viruses in the sub-array of that call
  - Since our recursion splits the input in half for each recursive call, the sizes of m_{left} and m_{right} will either be both $\frac{n}{2}$ when n is even, or in the case where n is odd, one of the "halves" will have $m = \frac{n}{2}$, and the other will have $m = \frac{n}{2} + 1$.
  - Then, we show that the maximum size of EC, based on the fact that it was not returned as the result of either side of the recursion, is strictly less than the minimum size required to be the "most prevalent" in the array of size n:

$$Let\ k \in Z^+$$

$$for\ n\ is\ even, n = 2k: \quad \frac{m_{left}}{2} + \frac{m_{right}}{2} = \frac{\left(\frac{n}{2}\right)}{2} + \frac{\left(\frac{n}{2}\right)}{2} = \frac{\left(\frac{2k}{2}\right)}{2} + \frac{\left(\frac{2k}{2}\right)}{2} = k$$

$$k\ < \frac{n}{2} + 1 = \frac{2k}{2} + 1 = k + 1$$

$$for\ n\ is\ odd, n = 2k + 1: \quad \frac{m_{left}}{2} + \frac{m_{right}}{2} = \frac{\left(\frac{n}{2}\right)}{2} + \frac{\left(\frac{n}{2} + 1\right)}{2} = \frac{\left(\frac{2k+1}{2}\right)}{2} + \frac{\left(\frac{2k+1}{2} + 1\right)}{2}$$

$$= \frac{k}{2} + \frac{k+1}{2} \ (by\ Lemma\ 1a)$$

$$= k\ (by\ Lemma\ 1b)$$

$$k < \frac{n}{2} + 1 = \frac{2k+1}{2} + 1 = k + 1\ (Lemma\ 1a)$$

- Therefore, if EC was not returned by either recursive call to the sub-arrays of size m, it is not a candidate for the "most prevalent" in the array of size n
- This leaves us with only 2 possible candidates for "most prevalent" in the array of size n, namely the returned equivalence class from the left sub-array and that of the right sub-array.

➢ Finally, we show that our iterative scan's with our 2 candidates will correctly identify whether one of our candidates is the "most prevalent" of the array of size n
- If the candidate is from side x, we scan side y and count the number of Viruses equivalent to the candidate, and arrive at a total by adding the size of the equivalence class contained in side x (which has been captured by the wrapper object returned from side x). If this total is at least $\frac{n}{2} + 1$, we simply update the count in the wrapper and pass it along up the recursion tree.
- The same can be said for candidate y
- If neither of these returned a "most prevalent," we have exhausted our candidates for "most prevalent" and thus return null (and at the top level of the recursion, we interpret this null as NOT_FOUND)

Code to Pseudo-code Mapping

My Java code essentially has a 1-1 mapping to the pseudo-code, with the following details:

- My wrapper object (to hold a Virus, its index, and the equivalence class size) is implemented as CanonicalVirus.java, with fields for Virus, index and count
- The "left-half" and "right-half" intervals are calculated by dividing the number of elements in the interval (hi – lo + 1) by 2, and using that as the "hi" of the left-recursion call interval, and using the value just above that as the "lo" of the right-recursion call interval
- I use a small-constant cutoff-to-brute-force of 6