Jason Caplan

Design and Analysis of Algorithms

Professor Leff

HW5

1. Vanilla binary search is used to find a specific, predetermined value in a fully sorted array. In our case, we are not looking for a predetermined value, rather we are looking for the median, whatever it may be. Additionally, our data is split into two arrays which, although sorted internally, are not sorted with respect to each other. More specifically, in a vanilla binary search, looking at the middle of the current interval under question will tell us in which half of the interval our target value is located, and we can then disregard the other half and recurse inward. However here, looking at the middle of a database does not obviously give us information about where the median might be.

2. We will take advantage of the fact that if we can show that a certain value or set of values has an imbalance between values greater than it and values less than it, then by definition it can be eliminated from consideration of being the median. And we can always eliminate $\sim\frac{1}{2}$ of the values by considering 2 intervals that collectively have a balance of values outside the intervals that are greater and less than the entire interval, and then looking at the middles of the intervals: Since values are distinct, we will have $mid_1 < mid_2$, and it follows that everything less than mid_1 has more values above it than below it and can be eliminated, and everything greater than mid_2 has more values below it than above it and can also be eliminated. This way we can "squeeze" the intervals until we find the median.

3. (see below)

Pseudo-code

```
# makes top-level call to recursive method
findMedian($db_a$, $db_b$):
    return find($db_a$[0 → end], $db_b$[0 → end])

# recursive method, eliminates half the problem size
find($interval_a$, $interval_b$):
    span = (length of interval) - 1
    if (span < small constant c):                                   } Base Case → O(1)
        # hand off to $O(1)$ brute-force                            }
        return solve($interval_a$, $interval_b$)                    }

    # $mid_a$ and $mid_b$ are true center for odd length intervals, and
    # for even, $mid_a$ is just left of center, $mid_b$ just right of center
    $mid_a$ = $start_a$ + (span / 2)                                } Iterative Work → O(1)
    $mid_b$ = $start_b$ + (span / 2)                                }
    if (length of interval is even) increment $mid_b$ by 1          }

    # recurse
    if ($db_a$.query($mid_a$) < $db_b$.query($mid_b$)):             } 1 recursive call,
        # eliminate left of $interval_a$ and right of $interval_b$  } halve the size of
        find($mid_a$→$high_a$, $low_b$→$mid_b$)                     } the data, so T(n/2)
    else:   # strictly greater because no duplicate values          }
        # eliminate right of $interval_a$ and left of $interval_b$  }
        find($low_a$→$mid_a$, $mid_b$→$high_b$)                     }

# $O(1)$ brute-force method to solve for median of small intervals
# of a predetermined constant length c (e.g. c=6)
solve($small\_interval_a$, $small\_interval_b$):
    # combine using the $O(n)$ "merge" step of mergesort            } Takes O(2c+1),
    combo = merge(small_intervala, $small\_interval_b$)             } so O(1)
    return combo[mid] # using smaller of 2 mid's (if even length)   }
```

Recurrence Relation

Each call to **find** consists of an $O(1)$ call to **solve** in the base case (since it takes $O(2c)$ to merge and O(1) to access the mid), otherwise $O(1)$ work calculating the mid values and comparing them, and a recursive call to with intervals that are half as large. (Technically the new interval size is $\frac{n}{2} + 1$ in the case of an odd-length interval, and either $\frac{n}{2}+1$ or $\frac{n}{2}$ in the case of an even-length interval, depending on whether **dbₐ.query(midₐ) < db_b.query(mid_b)** or **dbₐ.query(midₐ) > db_b.query(mid_b)** respectively. But from Big-O perspective, this extra 1 is insignificant). This yields the following recurrence for $Q(n)$:

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

Performance

Applying the Master Theorem to the aforementioned recurrence relation,

$$a = 1, \qquad b = 2, \qquad d = 0$$

$$1 = 2^0$$

$$Q(n) = n^0 \times \log(n) = \mathbf{log(n)} \ \blacksquare$$

Correctness

For this proof, "the balance invariant" means the following: given intervals from each database, interval$_a$ and interval$_b$, the number of accounts greater than every account in both intervals is equal to the number of accounts less than every ccount in both intervals. By definition, this ensures that the true global median lies within the two intervals.

- As long as each call to **find** maintains the invariant, we will not have excluded the median from the intervals when we finally call **solve**. Furthermore, **solve** identifying the local median of the combined intervals will correctly identify the global median. We prove this here:
    - First, let's show that **solve** works to find the local median its 2 input intervals:
        - Once we merge the two intervals, they are sorted, so the median will be located at the middle of that array (middle defined as the element at index $(\frac{arraylength}{2})$ for odd $arraylength$, and index $(\frac{arraylength}{2}) - 1$ for even $arraylength$)
    - Next we show that **solve** actually works to find the global median of the 2 databases:
        - The local median of the 2 input intervals, by definition, has the same number of accounts smaller than it as the number of accounts larger than it. But since we assume that **find** maintained the balance invariant, it is also true that globally, the number of accounts in both databases that are smaller than it is the same as the number of accounts in both databases that are larger than it. Therefore, it is indeed the global median.
- Now we just need to show that **find** maintains the balance invariant:
    - On the top-level call to **find**, where the intervals include are both whole databases, it holds:

$$(\#\ accounts < intervals) = (\#\ accounts > intervals) = 0$$

- In an induction style, we show that if it held for the previous call to find, it will hold to the next call to find, and therefore holds for all calls to find:
  - Call n the length of each interval respectively. There are 2 cases to consider, n is odd, and n is even:
    - For n is odd, mid$_a$ and mid$_b$ will be the true center account in each interval. For the next call to find, we eliminate exactly $\frac{n}{2}$ (Java integer division) accounts smaller than all accounts in both intervals of the next call to find, and exactly $\frac{n}{2}$ accounts larger than all accounts in both intervals of the next call to find. Since the number of accounts smaller than the new intervals is increasing by the same number as the number of accounts larger than the new intervals, the balance invariant is maintained for the new intervals.
    - For n is even, mid$_a$ will be just left of true center of interval$_a$, and mid$_b$ will be just right of true center of interval$_b$. There are 2 sub-cases:
      - If db$_a$.query(mid$_a$) < db$_b$.query(mid$_b$), then we eliminate exactly $\frac{n}{2} - 1$ accounts smaller than all accounts in both intervals of the next call to find, and exactly $\frac{n}{2} - 1$ accounts larger than all accounts in both intervals of the next call to find. By the same logic as above, the balance invariant is maintained.
      - If db$_a$.query(mid$_a$) > db$_b$.query(mid$_b$), then we eliminate exactly $\frac{n}{2}$ accounts smaller than all accounts in both intervals of the next call to find, and exactly $\frac{n}{2}$ accounts larger than all accounts in both intervals of the next call to find. By the same logic as above, the balance invariant is maintained. ■