CS208 · Algorithm Design and Analysis

Assignment 5

何泽安 (He Zean)
12011323
Dec. 5, 2022

Ch.5 - Ex.1

```
1  def find_nth_big(db1: Database, db2: Database, n: int) -> Numeric:
2      return helper(db1, db2, 0, 0, n)
3
4
5  def helper(db1: Database, db2: Database,
6              base1: int, base2: int,  # the part we have taken
7              p: int  # terms we still need to take to reach a half
8      ) -> Numeric:
9      """Returns the nth element of the union of elements in two databases.
10
11     Assumes that we have a table that merges db1 and db2, and sorts the elements
12     inside, then the [0 - base1]th elements in db1 and the [0 - base2]th elements
13     in db2 are considered smaller than the n-th element amone the 2n elements, aka.
14     they are the conquerd part and in the low half part of the merged-sorted table.
15
16     Args:
17         base1, base2: In the low half part of the 2n elements, we already know there are
18                       (at least) base1 elements from db1 and base2 elements from db2.
19         p: After conquerd base1 + base2 elements, we still need to conquer p elements to reach n.
20
21     Returns:
22         The n-th element of the union of db1 and db2 (2n elements in total).
23     """
24     if p == 1:  # base1 + base2 = n - 1, as conquered, we now need the last element as result
25         ea, eb = db1.query(base1 + 1), db2.query(base2 + 1)
26         return min(ea, eb)
27     ea, eb = db1.query(base1 + p // 2), db2.query(base2 + p // 2)
28     if ea < eb:  # we can safely take the next p/2 elements from db1 into 'low n elements'
29         return helper(db1, db2, base1 + p // 2, base2, p - p // 2)  # still we need p - p/2 elements
30     else:
31         return helper(db1, db2, base1, base2 + p // 2, p - p // 2)
```

That's quiet similar to Quiz 1 of DSAA, where we take *base1* elements (from small to large) in database DB_1 and *base2* elements in database DB_2 in hand during the process, where $\text{base1} + \text{base2} < n$ and it's possible to keep one of them be zero during the whole process. In each turn, if the k^{th} element in one database is greater than the other's k^{th} one, it's trivial that that element is greater then $k - 1$ elements in that database, and at least k elements in the other database, aka. that element ranks $2k$ or more in the total $2n$ elements.

Under this theorithm, we can conquer half of left problem each time, as line 26 shows. When we choose the part of database that has a smaller "biggest" element and place that $\frac{p}{2}$ elements into the selected table, this can promise that no element greater than that part is placed into the selected table, aka. in the whole process, we only select elements that are smaller then the n^{th} element among the two databases. Finally, when we reach the edge (base condition) that only one element is left to be chosen, we will select the smaller one among the two databases' left part, that will be the n^{th} element, which was what we want.

~~We then apply the master theorem to analyze its time complexity. It's trivial that for each task *helper(_, _, _, _, n)*, we can say its problem size is n. Then we can easily check line 29 and 31, under an *if* cluse, which means that this task is divided into one single sub-task, having a problem size $\frac{n}{2}$. Since we are actually finding out its queries time, not the actual time complex, we can consider the other part in a recursion takes $O(1)$, then we have $T(n) = 1 \cdot T(n/2) + O(1)$, that's an $O(\log n)$.~~ (Master theorem is more formal, but may not be so suitable for calculating "query times".)

Let $T(n)$ be the query time for getting the n^{th} element among two n sized databases. If $n > 1$, then two queries are applied, and a $T(n/2)$ is needed, otherwise, if $n = 1$, then we will have our last two queries. The below formula is enough to show that $T(n) = O(\log n)$.

$$T(n) = \begin{cases} 2 & (n = 1) \\ T(n/2) + 2 & (n > 1) \\ \text{impossible.} & (\text{otherwise}) \end{cases}$$

Ch.5 - Ex.2

```python
from typing import Tuple


def split(arr: list) -> Tuple[list, int]:
    """Returns the sorted array and the number of significant inversions."""
    if len(arr) <= 1:  # base case, no inversions can exist
        return arr, 0

    mi = len(arr) // 2  # cut from middle, split and to be conquerd
    left, l_sig_cnt = split(arr[:mi])
    right, r_sig_cnt = split(arr[mi:])
    merged, m_sig_cnt = merge(left, right)  # merge the two halves
    return merged, l_sig_cnt + r_sig_cnt + m_sig_cnt


def merge(lo: list, hi: list) -> Tuple[list, int]:
    """Merges two sorted lists and returns the merged list and the number of significant inversions.

    Args:
        lo, hi: two sorted arrays to be merged
    """
    res = []
    sig_cnt = 0
    l, h = 0, 0  # indices for lo and hi when comparing 'significant' and merging

    # after that, even if there lefts elements in lo or hi
    # they are already sorted and thus cannot be inversions
    while l < len(lo) and h < len(hi):
        if lo[l] > 2 * hi[h]:
            sig_cnt += 1
```

```python
                h += 1
            else:
                l += 1

        l, h = 0, 0
        while l < len(lo) and h < len(hi):
            if lo[l] < hi[h]:
                res.append(lo[l])
                l += 1
            else:
                res.append(hi[h])
                h += 1
        # one of lo or hi may left several elements, but be greater than all elements in
        # the other array, and also be sorted thus cannot contribute to #inversions
        res.extend(lo[l:])
        res.extend(hi[h:])

        return res, sig_cnt


if __name__ == '__main__':
    _, cnt = split([11, 2, 9, 5, 3])
    print(cnt)  # 4
```

This algorithm is almost the same of counting inversions, or the merge sort itself, but a counter of significant inversions is added. We can find the merge step updates the counter when lo[l] > 2 * hi[h], following the requirement, whose correctness is just like counting inversions (where lo[l] > hi[h]).

Also, assume that len(lo) = n, we then have len(hi) = n or n + 1; for the two *while* loop, each takes $\Theta(n)$, thus merge is $\Theta(n)$. Then, the split step is $T(n) = 2T(n/2) + \Theta(n/2) = 2T(n/2) + \Theta(n)$, having $\Theta(n) = \Theta(n^{\log_b a})$ where $a = b = 2$, we have $T(n) = \Theta(n \log n) = O(n \log n)$ by the master theorem.