COMP SCI 577 Homework 02 Problem 3

Divide and Conquer

Ruixuan Tu

rtu7@wisc.edu

University of Wisconsin-Madison

26 September 2022

Algorithm

Explanation

We have the main subroutine `fast_select` to get the pair (a_m, w_m) for $\sum_{a_i < a_m} w_i < k$ and $\sum_{a_i \le a_m} w_i \ge k$ from the inputs $A = [a_i]$, $W = [w_i]$, and k, with $k = \frac{1}{2}$ at initial. Denote $[0, n)$ be the range of $[i]$. The base case is that $n = 1$ so that there is nothing to select except for the only element. Then we calculate the array of medians A_m and W_m of the $n' = \lceil \frac{n}{w} \rceil$ with $w = 5$ consecutive length-w segments of A and W by the subroutine call `get_median_arr(A, W, 5)`. By the inspiration from the lecture, we want to get the median of the medians, i.e., the median of A_m and W_m; and this is done by the subroutine call `fast_select(compress(A_m, W_m), sum(W_m) / 2)`. We use the median of medians as the pivot as in the lecture. After that, we split A and W by the pivot A_p, just as in the quick sort, to get the left part A_L and W_L in L; the middle part in M; and the right part in R. We know from the `split` subroutine that $\sum_{a_i < a_m} w_i = \sum W_{L_i}$ and $\sum_{a_i \le a_m} w_i = \sum W_{L_i} + \sum W_{M_i}$ with the needed sums in variables `L_Wsum` and `M_Wsum`. If the pivot p satisfies the two conditions of k, then p is the correct median of this subroutine without further calculation. If not, then the pivot is either too low or too high, and we need to search in the inverse interval (i.e., search in right if too low, or left if too high). While searching in right, we need to have new $k' = k - \sum W_{L_i} - \sum W_{M_i}$ to ensure that we do not want to find the sum of weights that are not in the right in the search.

The subroutines `compress` and `extract` are compressing/extracting lists A and W to/from one list of tuples $[(A_i, W_i)]$ for code readability. Hence, the subroutines `split` and `get_median_arr` are almost identical to the versions introduced in the lecture, with `get_median` be a helper function.

Code (Python)

```python
from math import ceil
from typing import List, Tuple

def readline_floats(s: str) -> List[float]:
    return list(float(x) for x in s.split(" "))

def compress(A: List[float], W: List[float]) -> List[Tuple[float, float]]:
    return list(zip(A, W))

def extract(D: List[Tuple[float, float]]) -> Tuple[List[float], List[float]]:
    A: List[float] = [x for x, _ in D]
    W: List[float] = [x for _, x in D]
    return A, W

def split(D: List[Tuple[float, float]], p: float) -> Tuple[List[Tuple[float, float]], List[Tuple[float, float]], List[Tuple[float, float]], float, float, float]:
    A, W = extract(D)
    L: List[Tuple[float, float]] = []
    M: List[Tuple[float, float]] = []
    R: List[Tuple[float, float]] = []
    L_Wsum: float = 0.0
    M_Wsum: float = 0.0
    R_Wsum: float = 0.0
    for i in range(len(A)):
        if A[i] < p:
            L.append((A[i], W[i]))
            L_Wsum += W[i]
        elif A[i] > p:
            R.append((A[i], W[i]))
            R_Wsum += W[i]
        else:
```



```python
            M.append((A[i], W[i]))
            M_Wsum += W[i]
    return L, M, R, L_Wsum, M_Wsum, R_Wsum

def get_median(D: List[Tuple[float, float]]) -> Tuple[float, float]:
    S: List[Tuple[float, float]] = sorted(D, key=lambda x: x[0])
    A_s, W_s = extract(S)
    W_Lsum: float = 0.0
    W_scale: float = sum(W_s)
    for i in range(len(D)):
        if W_Lsum < 0.5 * W_scale and W_Lsum + W_s[i] >= 0.5 * W_scale:
            return (A_s[i], W_s[i])
        W_Lsum += W_s[i]
    return (None, None)

def get_median_arr(A: List[float], W: List[float], w: int) -> Tuple[List[float], List[float]]:
    n_A: int = len(A)
    n_median_arr: int = ceil(n_A / w)
    M_A_splited: List[float] = []
    M_A_merged: List[float] = []
    M_W_splited: List[float] = []
    M_W_merged: List[float] = []
    for i in range(n_median_arr):
        m_A: List[float] = []
        m_W: List[float] = []
        r: int = w
        if i == n_median_arr - 1 and n_A % w != 0:
            r = n_A % w
        for j in range(r):
            m_A.append(A[i * w + j])
            m_W.append(W[i * w + j])
        M_A_splited.append(m_A)
```



```python
        M_W_splited.append(m_W)
    for i in range(n_median_arr):
        m_A, m_W = get_median(compress(M_A_splited[i], M_W_splited[i]))
        M_A_merged.append(m_A)
        M_W_merged.append(m_W)
    return M_A_merged, M_W_merged

def fast_select(D: List[Tuple[float, float]], k: float) -> Tuple[float, float]:
    A, W = extract(D)
    n = len(A)
    if n == 1:
        return (A[0], W[0])
    A_m, W_m = get_median_arr(A, W, 5)
    p = fast_select(compress(A_m, W_m), sum(W_m) / 2)
    L, _, R, L_Wsum, M_Wsum, _ = split(compress(A, W), p[0])
    L_A, L_W = extract(L)
    R_A, R_W = extract(R)
    if L_Wsum < k and L_Wsum + M_Wsum >= k:
        return p
    elif L_Wsum < k:
        return fast_select(compress(R_A, R_W), k - L_Wsum - M_Wsum)
    else: # L_Wsum >= k
        return fast_select(compress(L_A, L_W), k)

if __name__ == "__main__":
    F = open("hw02.in", "r")
    L = F.readlines()
    A = readline_floats(L[0]) # 40 -5 4 0 2.5 6 -2
    W = readline_floats(L[1]) # .25 .1 .05 .18 .15 .2 .07
    print(fast_select(compress(A, W), sum(W) / 2)[0])
```

Correctness

Induction

Claim: The algorithm is correct for any array with a size of n with $n \in \mathbb{N}^+$.
Base Case: $n = 1$. There is nothing to select except for the only element.
Inductive Step: Assume that the algorithm is correct for $n = j$ for all $j < s$ by strong induction. For $n = s$, we have an approximate pivot A_p which is the guaranteed median of medians, and we have proved in the lecture that it is the ρ-approximate median with $\rho = \frac{3}{4}$ of A with weights in W. Then we do the proof by cases with A_m being the true median at some percentile indicated by the parameter k.

For $A_p = A_m$, we have already done with the result, and it could be verified by `split` as we have mentioned in Explanation. For $A_p < A_m$, the `split` function would check for the wrong median which violates the condition $\sum_{a_i \le a_m} w_i = \sum W_{L_i} + \sum W_{M_i} \ge k$, and we can get the correct result by calling the subroutine as described in Explanation which is guaranteed by the induction hypothesis. For $A_p > A_m$, the `split` function would check for the wrong median which violates the condition $\sum_{a_i < a_m} w_i = \sum W_{L_i} < k$, and similarly, we can get the correct result by calling the subroutine as described in Explanation which is guaranteed by the induction hypothesis.
Note: The helper subroutine `get_median` is not affected by $\sum W_i \neq 1$ as the sum of weights are normalized by $W_{\text{scale}} = \sum W_i$ that should compare with $0.5 * W_{\text{scale}}$ instead of just 0.5. The main subroutine `fast_select` is also not affected, as we have used percentile k instead of 0.5 to split the array, which could also solve the searching problem in the right part as discussed in Explanation. Also, the situation with duplicate values of pivot is considered that $\sum W_{M_i}$ is introduced to calculate all weights of the current pivot value.

Termination

The algorithm is diminishing the problem size, as after executing `split`, there must be at least 1 element (i.e., the median) excluded from the original list, as we could only continue searching in the left sub-list or the right sub-list. Hence, the base case should cover all $n \in \mathbb{N}^+$, as from the base case, there must be at least one $|A_L| > 0$ or $|A_R| > 0$, and then we can continue to split in the middle in the other part even if one of $|A_L$ or A_R is empty.

Complexity

The helper subroutine `get_median` has the time complexity of $O(1)$, as there is always $n = w = 5$ tuples to sort with one time of $n < w = 5$ tuples. The complexities of `get_median_arr` and `split` are both $O(n)$ as introduced in the lecture without the increase in the most significant term in complexity from the implementation.

For the main subroutine `fast_select`, from Master Theorem (at Page 11 of this UCSD slide), we have the recurrence of `fast_select` in the form $T(n) = T(\frac{3n}{4}) + 2n$ in the worst case with $2n \in O(n)$, then with $a = 1$ and $b^d = (\frac{4}{3})^1 = \frac{4}{3} > a$, $T(n) \in O(n^1) = O(n)$. The proof of both $\frac{3}{4}$ and the complexity of $O(n)$ with $w \ge 5$ in `fast_select` is introduced in the lecture, as well as that the structure of `fast_select` does not change in complexity (i.e., there is no new call to any function).

Appendix

Code (Python) of $O(n \log(n))$ for Reference

```python
from typing import List, Tuple

def readline_floats() -> List[float]:
    return list(float(x) for x in input().split(" "))

def solve(A: List[float], W: List[float]) -> float:
    S: List[Tuple[float, float]] = sorted(zip(A, W), key=lambda x: x[0])
    A_s: List[float] = [x for x, _ in S]
    W_s: List[float] = [x for _, x in S]
    W_scale = sum(W)
    W_sum: float = 0.0
    for i in range(len(A)):
        if W_sum < 0.5 * W_scale and W_sum + W_s[i] >= 0.5 * W_scale:
            return A_s[i]
        W_sum += W_s[i]
    return -1

```



```python
if __name__ == "__main__":
    A = readline_floats() # 40 -5 4 0 2.5 6 -2
    W = readline_floats() # .25 .1 .05 .18 .15 .2 .07
    print(solve(A, W))
```

Code (Python) for Test Cases Generation

```python
import cyaron
import correct # reference program above
import numpy as np

if __name__ == "__main__":
    for data_id in range(2):
        io = cyaron.IO("test" + str(data_id) + ".in", "test" + str(data_id) + ".ans")
        n = cyaron.randint(1, 100)
        W_raw = np.random.rand(n).tolist()
        W_sum = sum(W_raw)
        W = [w / W_sum for w in W_raw]
        A = np.random.randint(-100, 100, size=n).tolist()
        io.input_writeln(A)
        io.input_writeln(W)
        io.output_writeln(correct.solve(A, W))
```

Test Cases

Input: $A \leftarrow [40.0, -5.0, 4.0, 0.0, 2.5, 6.0, -2.0]$; $W \leftarrow [0.25, 0.1, 0.05, 0.18, 0.15, 0.2, 0.07]$
Output: 2.5

Input: $A \leftarrow$ [-62.0, -59.0, -80.0, 64.0, -93.0, -11.0, -70.0, 2.0, -99.0, 60.0, -65.0, 22.0, 7.0, 77.0, -75.0, 75.0, -45.0, -45.0, 42.0, -49.0, 95.0, 54.0, 54.0, 31.0, 2.0, -32.0, -37.0, -70.0, -20.0, 31.0, -3.0, -20.0, 92.0, 99.0, 34.0, 89.0, 66.0, 36.0, 99.0, -7.0, 83.0, -31.0, 75.0, -50.0, 27.0, -89.0, -93.0, -39.0, -28.0, -13.0, -15.0, -80.0, 4.0, -19.0];
$W \leftarrow$ [0.009635088510133326, 0.01730005414042247, 0.02205677079879268, 0.020377264777550056, 0.023309273988243534, 0.0020798175336173234, 0.00646417831224761, 0.01440650755439465, 0.029622716607202865, 0.005088972287934939, 0.02917462263618285, 0.009359093492039219, 0.02840509054872456, 0.0034666670373809183, 0.017105965508466214, 0.007954253637394796, 0.010931245263376382, 0.024797288149057933, 0.02927029514042684, 0.014964927952948305, 0.030544887305412625, 0.0061719533271399965, 0.028780346395749192, 0.014494666008948831, 0.004884289444556541, 0.022578017782974833, 0.01876316887585921, 0.01999550468439874, 0.0023745843458433027, 0.03436804250114482, 0.014978248692100479, 0.03304836667207052, 0.007752614318740494, 0.017687473374281594, 0.015500061550499624, 0.010520516206031446, 0.030807729492458854, 0.013576018263922247, 0.02759272950897091, 0.02913873838234767, 0.012867570369263876, 0.022035106390367726, 0.003634433211453344, 0.025695598581046, 0.03203468856922726, 0.018508760024377784, 0.03646994329808107, 0.025424496112484707, 0.022800855532513948, 0.008121468814818945, 0.01168070992222032, 0.011188343835430242, 0.02763261403225525, 0.03257736029646955]
Output: -15.0