CS577: Homework 3

Haruki Yamaguchi
hy@cs.wisc.edu

Keith Funkhouser
wfunkhouser@cs.wisc.edu

October 1st, 2015

1 Introduction

The algorithm is given below, with Algorithm 1 containing the main functions and Algorithm 2 containing the "auxiliary" functions which assist in the computation.

The algorithm utilizes a divide-and-conquer technique similar to the one shown for the selection algorithm shown in class [1]. At each step, the sequence a (and, correspondingly, w, which is kept "in line" with a such that any time a swap in a is made, the same swap in w is made):

1. has its median computed
2. is partitioned on that median, such that all elements to the left (L) are smaller and all elements to the right (R) are at least as large
3. has the weights of the elements in L summed

Depending on the sum of the weights in L and w_{pivot}, we can determine whether to recurse on L, R, or simply return. This gives an overall $O(n)$ runtime due to the halving of the problem size at each step.

2 Algorithm

Algorithm 1: Weighted median algorithm

`//NOTE: The problem description is 1-indexed, i.e. the sequence is given as` $a_1, a_2, \ldots, a_n$. `Here, WLOG we refer to the 0-indexed version` $a_0, a_1, \ldots, a_{n-1}$ `for simplicity of implementation in Python.`

Input: A sequence of n real numbers $a_0, a_1, \ldots, a_{n-1}$ and a corresponding sequence of non-negative real weights $w_0, w_1, \ldots, w_{n-1}$ such that $\Sigma_{i=0}^{n-1} w_i = 1$.

Output: The weighted median of the sequence, a_k, such that $\Sigma_{a_i < a_k} w_i < 0.5$ and $\Sigma_{a_i \leq a_k} w_i \geq 0.5$.

`WeightedMedian(`a,w`):`
    return `WeightedMedianRec(`*a, w, 0, n-1, 0.5*`)`

Input: The same sequences of numbers a and weights w as above, in addition to two non-negative integers lo and hi, $0 \leq lo \leq hi < n$, and a $target \in (0, 1]$ which is the "target" for the weighted median, e.g. 0.5 in the problem description.

Output: The weighted "median" (in quotes because $target$ may not be 0.5) of the sequence $a_{lo}, \ldots, a_{hi}$, such that $\Sigma_{a_{lo} \leq a_i < a_k} w_i < target$ and $\Sigma_{a_{lo} \leq a_i \leq a_k} w_i \geq target$.

`WeightedMedianRec(`*a, w, lo, hi, target*`):`
    if *lo=hi* **then**
        return a_{lo}
    else
        a, pivot ← `Median(`*a,lo,hi*`)`
        a, w, pivot ← `Partition(`*a,w,lo,hi,pivot*`)`
        total ← `SumWeights(`*w,lo,pivot*`)`
        if *total >target* **then**
            return `WeightedMedianRec(`*a,w,lo,pivot - 1,target*`)`
        else
            if w_{pivot} + *total* ≥ *target* **then**
                return a_{pivot}
            else
                return `WeightedMedianRec(`*a,w,pivot + 1,hi,target - total -* w_{pivot}`)`
            end
        end
    end

Algorithm 2: auxiliary functions

Input: The same sequence of numbers a as before, in addition to two non-negative integers lo and hi, $0 \leq lo \leq hi < n$.

Output: a and $pivot$, the index of the median of of $a_{lo}, \ldots, a_{hi}$, such that $lo \leq return \leq hi$.

`Median(`a,lo,hi`)`:

    //Uses a variant of the median-of-median algorithm described in the Divide and Conquer course notes [1, pp. 28-32]. The new version should also take a low bound lo and high bound hi, and return the index of the median within those bounds (inclusive). This runs in O(n) worst-case time. We omit the pseudocode here as the changes are only minor and do not change the overall structure of the code. For reference, see the SELECT algorithm described in Cormen, et al. pp.220-221.

Input: The sequences a and w as above, in addition to bounds lo and hi, $0 \leq lo \leq hi < n$, and $pivot$, the index for the value of a around which the arrays a and w should be partitioned.

Output: a and w are sorted in place with respect to each other (i.e. any time a swap is made in a, it is also made in w. a and w are returned along with $pivot$, the index of the pivot value in a after the partitioning has finished.

`Partition(`$a,w,lo,hi,pivot$`)`:

    `//This is a modified version of the Partition subroutine called by QUICKSORT [2, p171], which rearranges the subsequences` $a_{lo}, \ldots, a_{hi}$ `and` $w_{lo}, \ldots, w_{hi}$ `in place such that` $\forall i \in [lo, pivot)$, $a_i < a_{pivot}$ `and` $\forall j \in [pivot, hi]$, $a_j \geq a_{pivot}$. `The returned value` $pivot$ `is the index of the pivot in` a `when finished.`

    pivotValue $\leftarrow a_{pivot}$
    swap a_{hi} with a_{pivot}
    swap w_{hi} with w_{pivot}
    i $\leftarrow$ lo
    for $j \leftarrow lo$ *to* hi *- 1* **do**
        if a_j $<pivotValue$ **then**
            swap a_i with a_i
            swap w_i with w_j
            i $\leftarrow$ i + 1
        end
    end
    swap a_i with a_{hi}
    swap w_i with w_{hi}
    return a, w, i

Input: w, as defined previously, and two indices of w, $0 \leq lo \leq pivot < n$.

Output: The sum of the weights $w_{lo} + \cdots + w_{pivot-1}$.

`SumWeights(`$w,lo,pivot$`)`:
    total $\leftarrow 0$
    for $i \leftarrow lo$ *to* $pivot$ *- 1* **do**
        total $\leftarrow$ total + w_i
    end
    return total

3 Correctness

3.1 `Partition` and `Median`

For the sake of being brief we will omit the correctness arguments for the auxiliary functions `Median` and `Partition` and let it suffice to say that they will terminate with correct values assuming valid inputs. The `Partition` method functions exactly as the algorithm given in Cormen, et al. p. 171, apart from swapping the value in both a and w instead of just a, and first swapping the *pivot* value with the *hi* value.

3.2 `SumWeights`

Consider the loop invariants:

$$i \in \mathbb{Z} \tag{1}$$

$$i \leq pivot \tag{2}$$

$$total = \sum_{j=lo}^{i-1} w_j \tag{3}$$

The proof of these invariants follows by induction. The first time the loop is executed, $total = 0$ and $i = lo$. (1), (2), and (3) all hold trivially. Assume that when the loop condition ($i < pivot$) is tested for the $(t+1)$st time, invariants (1), (2), and (3) held the tth time the loop condition was tested. Since the loop condition evaluated to true (i.e. $i < pivot$) on the tth time (since we have reached the $(t+1)$st time), the body of the loop executed. After execution, we have $i_{t+1} = i_t + 1 \leq pivot$ by (1) and the loop condition of the previous iteration, and $i_{t+1} = i_t + 1 \in \mathbb{Z}$ by closure of addition of integers. Finally, (3) holds because $total_{t+1} = total_t + w_{t+1} = \left(\sum_{j=lo}^{i_t-1} w_j\right) + w_{i_{t+1}} = \left(\sum_{j=lo}^{i_t} w_j\right) = \left(\sum_{j=lo}^{i_{t+1}-1} w_j\right)$, by the inductive hypothesis.

When the loop does halt, we know that the loop condition is false, i.e. $i \geq pivot$, which combined with (2) gives $i = pivot$. Hence the output $total = \sum_{j=lo}^{i-1} w_j = \sum_{j=lo}^{pivot-1} w_j$ as desired.

At every loop in the iteration, the counter i is incremented, and it is initialized at $i = lo \leq pivot$, so eventually it will be at least as large as *pivot* and the loop will halt.

3.3 `WeightedMedian` and `WeightedMedianRec`

Assuming valid inputs for `WeightedMedian`, it has only one function call and depends on the termination and correct return value of `WeightedMedianRec`, which we show below.

3.4 Partial correctness

Partial correctness follows if, for any valid set of inputs, both of the following are true [3]:

- all the recursive calls that appear in the code of the program on that input have valid arguments
- assuming all those calls return a correct output for their respective arguments, and that the program terminates on that input, the program returns a correct output on that input

The first call to `WeightedMedianRec` certainly contains valid inputs, since a and w are valid inputs to `WeightedMedian`, $0 \leq lo \leq hi < n$, and $target \in (0, 1]$. In the base case, $lo = hi$ and we return the value. Otherwise, we have $lo \neq hi \Rightarrow lo < hi$ by the precondition $0 \leq lo \leq pivot \leq hi < n$.

The call to `Median` is valid by the condition $0 \leq lo < hi < n$, and the return value of `Median` is assumed to be correct: $pivot \in [lo, hi]$. Then, the inputs to `Partition` are valid, since $0 \leq lo \leq pivot \leq hi < n$. Assuming that `Partition` performs the correct operations and returns the correct valud $pivot \in [lo, hi]$, the inputs to `SumWeights` will be valid: $0 \leq lo \leq pivot < n$.

There are now two cases to consider:

1. Strict inequality: $0 \leq lo < pivot < hi < n$.
   If *pivot* is between *lo* and *hi*, then the calls to `WeightedMedianRec` are certainly valid, either:

- `WeightedMedianRec(`a,w,lo,$pivot$ - 1,$target$`)` which clearly has valid inputs $0 \leq lo \leq hi < n$ and the same a, w, and $target$
- `WeightedMedianRec(`a,w,$pivot$ + 1,hi,$target$ - $total$ - w_{pivot}`)`, which also has valid inputs $0 \leq lo \leq hi < n$, the same a and w. Since this recursive call is only made when $w_{pivot} + total < target$ and is made on $target = target - total - w_{pivot} > 0$, $target$ is also a valid input.

2. Equality: either $pivot = lo$ or $pivot = hi$ (but not both by the base case condition evaluating to false).

   - $pivot = lo$: Then, $total = \sum_{j=lo}^{pivot-1} w_j = \sum_{j=lo}^{lo-1} w_j = 0$. So we will reach the else branch on line 12 of algorithm 1, either returning a value or, if $w_{pivot} < target$, recurring on $lo = pivot + 1$, $hi = hi$ which is certainly a valid input.
   - $pivot = hi$: Then, we can never reach the recursive call on line 16 of Algorithm 1 because $w_{pivot} + total = w_{pivot} + \sum_{j=lo}^{pivot-1} w_j = \sum_{j=lo}^{pivot} w_j = \sum_{j=lo}^{hi} w_j$ which must necessarily be at least $target$, otherwise there would be valid inputs. Hence, we will either return a value in this case, or recur on $lo = lo$, $hi = pivot - 1$, which once again are valid inputs.

Since we have shown that the inputs for all calls to `WeightedMedianRec` are valid, we now move to show that, assuming those calls return a correct output and that the program terminates on that input, the program returns a correct output. The base case of the recursive function returns correctly trivially, i.e. for a sequence of length 1 where $lo = hi$, the first and only value must be the value a_k for which $\Sigma_{a_i < a_k} w_i < 0.5$ and $\Sigma_{a_i \leq a_k} w_i \geq 0.5$. The rest of the proof follows from the intuition about the problem given in the introduction. Namely, the sequence a is split into 3 subsequences: L, $pivot$, and R, where the length of $pivot$ is 1.

If the sum of the weights in L is at least as large as the $target$, which is 0.5 for the first invocation, then certainly the weighted median is located somewhere there, and we can assume that the recursive call on L with the same $target$ size returns the correct value. If, however, the sum of weights in L is smaller than the $target$, we have two cases:

1. if the weight of the pivot pushes the total "over the edge", i.e. $total + w_{pivot} > target$, then the pivot value is our solution and we return the correct value immediately.
2. if that is not the case, then we know that all elements in L as well as the pivot itself are below the weighted median. Furthermore, we know that all of those weights can be counted towards the $target$, and we can assume that the recursive call on R with a new $target$ that is decremented by the total weight of $L + w_{pivot}$ will return the correct value.

This concludes the proof for partial correctness, and we now aim to show that the program terminates for any valid input.

3.5 Termination

There does not exist a valid set of inputs for which the recursion tree will not "bottom out". This becomes clear when looking at the recursive calls in lines 11 and 16 of Algorithm 1, which both decrease the problem size (either reducing the search to $[lo, pivot - 1]$ or $[pivot + 1, hi]$. Furthermore, the median is guaranteed to partition the sequence a optimally, such that half of its elements will be to the left and half to the right. Thus, the problem is reduced in size with each recursive call, and there cannot be an infinite sequence of calls on any valid input.

4 Runtime

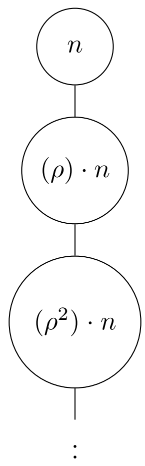
Each of the three steps described in the introduction, which correspond to lines 7, 8, and 9 of Algorithm 1, are completed in $O(n)$ time:

1. median computation: this was shown in the course notes to run in $O(n)$ worst-case time [1, p. 31]

2. sequence partitioning: `Partition` simply visits every element on $[lo, hi]$ once and does a constant amount of work each time (swapping values). Thus, it runs in $O(n)$ time.

3. weight summation: `SumWeights` simply visits every element on $[lo, pivot]$ once and does a constant amount of work each time (addition). Thus, it runs in $O(n)$ time.

Thus, the amount of work done locally at each level in the recursion tree is $c \cdot n$ for some c. Furthermore, a recursive call is only made on either L or R, which are each roughly half the size of the original sequence. So, the recursion tree looks as follows:



Since the work done at each node in the tree is linear, there is a constant c such that the work done at a node of size s is bounded by $c \cdot s$. Thus, at each level d of the recursion tree, at most $c \cdot \rho^2 \cdot n$ work is done. The total work, then, done over all levels of the recursion tree is:

$$c \cdot \left(1 + \rho + \rho^2 + \cdots\right) \cdot n$$

Taking the middle term out to infinity and for $|\rho| < 1$, we have:

$$\sum_{k=0}^{\infty} \rho^k = \frac{1}{1 - \rho}$$

Since $\rho = \frac{1}{2}$ because the size of the problem is divided in two at each level in the recursion tree, our runtime converges to $c \cdot \left(\frac{1}{1-\frac{1}{2}}\right) \cdot n = c' \cdot n = O(n)$ for some c'.

5 Optimizations

The algorithm could be optimized to be more more intimately involved with the approximate median algorithm described in class. Namely, it can be shown that finding the approximate median of a on $[lo, hi]$ is sufficient for reducing the problem size by $\rho \geq \frac{1}{4}$, and the runtime will still be $O(n)$. For simplicity, here we have described an algorithm that allows the approximate median algorithm to run to completion and return the true median, which is $O(n)$ but with a very large constant c in the runtime $c \cdot n$.

6 Python implementation

6.1 Algorithm code

```
import math

#for comparison of (in)equality with reals
tol = 0.000001

def weightedMedian(A,W):
  return weightedMedianRec(A,W,0,len(A)-1,0.5)

def weightedMedianRec(A,W,lo,hi,target):
  if(lo == hi):
    return A[lo]
  else:
    pivot = lowMedian(A,lo,hi)
    pivot = partition(A,W,lo,hi,pivot)
    total = sumWeights(A,W,lo,pivot)
    #total > target, recurse on L
    if total + tol > target:
      return weightedMedianRec(A,W,lo,pivot - 1,target)
    #total <= target
    else:
      #total + W[pivot] >= target, return pivot value
      if tol  >= target - (W[pivot] + total):
        return A[pivot]
      #total + W[pivot] < target, recurse on R
      else:
        return weightedMedianRec(A,W,pivot + 1,hi,target - total - W[pivot])

def sumWeights(A,W,lo,pivot):
  total = 0
  for i in range(lo,pivot):
    total += W[i]
  return total

#hacked implementation of O(n log n) median implementation
# NOTE we are suggesting that this be replaced with the worst-case O(n)
# median-of-medians algorithm discussed in class
def lowMedian(A,lo,hi):
  B = A[lo:hi+1]
  C = B[:]
  B.sort()
  # location of low median in
  k = int(math.ceil(len(B)/2.0)) - 1
  #return A.index(B[k])
  return lo + C.index(B[k])

#partition s.t. A[i] >= A[pivot] for all i >= pivot
# and A[i] < A[pivot] for all i < pivot
# see Cormen, et al. p. 171
def partition(A,W,lo,hi,pivot):
  pivotVal = A[pivot]
  swap(A,hi,pivot)
  swap(W,hi,pivot)
  i = lo
  for j in range(lo,hi):
```

```python
        if( A[j] < pivotVal ):
            swap(A,i,j)
            swap(W,i,j)
            i += 1
    swap(A,hi,i)
    swap(W,hi,i)
    return i

def swap(A,i,j):
    A[i],A[j] = A[j],A[i]
```

6.2 Testing

```python
from hw3 import weightedMedian, tol
from generateInput import generateInput

#the "true" weighted median, i.e. brute forced
def bruteForce(a,w,target=0.5):
    [sortA,sortW] = zip(*sorted(zip(a,w)))
    total = 0
    ptr = 0
    #while( total < target )
    while(tol < target - total):
        total += sortW[ptr]
        ptr += 1
    return sortA[ptr - 1]


##########################################
#              TEST CASES                #
##########################################

# sample size of 1000
N = 1000
for i in range(1,N+1):
    # generateInput returns a 2-tuple of values a and weights w, such that the
    # weights add up to 1. e.g.:
    #   >>> generateInput(4)
    #   [[-48889.55578699512, -17514.74434577306, -14318.16962721838,
    #   49178.17800571047], [0.366, 0.134, 0.39, 0.11]]
    [a,w] = generateInput(i)
    a_orig = a[:]
    w_orig = w[:]
    wm = weightedMedian(a,w)
    bwm = bruteForce(a,w)
    if wm != bwm:
        print("Brute force produced %d; you produced %d.\n A: %s\n W: %s" %
              (bwm,wm,str(a_orig),str(w_orig)))
print("Done testing!")
```

```
keith@keith-x220:~/code/cs/cs577/hw3$ python hw3_testing.py
Done testing!
```

References

[1] Dieter van Melkebeek. Divide and conquer, September 2015.

[2] Thomas H Cormen. *Introduction to algorithms*. MIT press, 2009.

[3] Dieter van Melkebeek. Program correctness, August-September 2015.