[lemma]Algorithm

Final Exam
Introduction to Algorithms
601.433/633
Spring 2020

Due: Friday, May 8th, 10am

Ethics Statement

I agree to complete this exam without unauthorized assistance from any person, materials, or device.

Name: Mou Zhang

Signature Date

1 Problem #1: YES/NO QUESTIONS (100 points) (10 questions, 10 points per question)

For every question answer yes or no.

1. Let T be an MST in graph $G = (V, E)$ and let $a, b \in V$ be two vertices. Let P be the path in T from a to b. P is always a shortest path from a to b.

   yes   no ✓

2. Prim and Kruskal algorithm may yield different spanning trees.

   yes ✓   no

3. There exist two strictly positive functions f, g such that $f(x) = o(g(x))$ and $(g(x))^2 = o((f(x))^3)$.

   yes ✓   no

4. If $f(n) = \Theta(g(n))$ then $2^{f(n)} = \Theta\left(2^{g(n)}\right)$.

   yes   no ✓

5. It is always possible to sort an array A of n real numbers in linear time if $\max_{i=1}^{n} |A[i]| \leq 5$.

   yes   no ✓

6. Let $G = (V, E)$ be a directed weighted graph and let $\delta(a, b)$ be the length of a shortest path from a to b. Let $a, b, c \in V$. If $\delta(a, b) + \delta(b, c) \leq \delta(a, c)$

then b belongs to a shortest path from a to c.

yes ✓ no

7. There is some fixed input X for which Randomized QuickSort always runs in $\Theta(n^2)$ time.

yes no ✓

8. For all undirected, weighted graphs, if we increase all weights by exactly one then all shortest paths do not change.

yes no ✓

9. An algorithm whose running time satisfies the recurrence

$$P(n) = 10P(n/2) + O(n^{10})$$

is asymptotically faster than an algorithm whose running time satisfies the recurrence

$$E(n) = 1.1E(n - 1000) + O(1).$$

yes ✓ no

10. Let G be undirected, weighted graph. If MST is unique then all weights are distinct.

yes no ✓

2 Problem #2: Basic (60 points)

You are given array A of n distinct integers. It is known that A has been sorted and then cyclically shifted (to the right) by k elements where k is unknown to you. For example, array $A = \{5, 6, 1, 2, 3, 4\}$ is sorted and cyclically shifted by $k = 2$ elements.

Design an algorithm for finding the median value of the elements in the array A. Prove the correctness of your algorithm and give the running time. Your answers must be explained clearly, and with enough details. The full credit will be given if the running time of your algorithm is $o(\sqrt{n})$ (little o).

Solution:

Algorithm 1 Find Median

```
procedure FIND MEDIAN($A, n$)
    $k \leftarrow Find\ k(A)$
    return A[(k + (n / 2)) mod n]   ▷ We use lower median for even length array
end procedure
procedure FIND K($A, n$)
    if $A[0] < A[n-1]$ then
        return 0                    ▷ the array is cyclically shifted by k = 0
    end if
    $left \leftarrow 0, right \leftarrow n-1$
    while $left < right$ do
        $mid \leftarrow (left + right)/2$
        if $A[mid] > A[n-1]$ then
            $left \leftarrow mid$
        else
            $right \leftarrow mid$
        end if
    end while
    return left
end procedure
```

Proof of correctness:

Proof. It is obviously very easy to find the median in a sorted array. We just need to find the number in the middle of the array, which is typically the $\frac{n}{2}$th element. This takes O(1) time. So now what we need to do is to find the *k* by which the array shifted. If k = 0, it's the same problem as not shifted, which is talked above. For k $\neq$ 0, since all numbers in the whole array is distinct and in order except the shifted place, there is only at one place p that $A[p] > A[p+1]$. And this place is the place where the array is shifted. Since it's the only place p that $A[p] > A[p+1]$, this also means that A[p] is the largest value in the whole array. On the left of place p, the numbers are in increasing order; on the right of place p, the numbers are in increasing order too. So it is easy to find the place p by binary search because of the following reason.

Since all numbers in A[0] to A[p] are larger than A[n - 1](the last number), in binary search, if A[mid] is bigger than A[n - 1], than it means current mid is on the left of p(including p), otherwise it's on the right side of p. Therefore, using the binary search in lines 13-21, we can find where the p is, then find out the value of k.

After that, since we have found the value of k, we can simply use the conditions to calculate the median in a sorted array. The median for array is $A[(k+\frac{n}{2})modn]$(for even length array, we only use lower median here for simplicity). This is because that comparing the old index q_{old} and the new index q_{new}, we can find out that $q_{new} = (q_{old} + k)modn$. This is because after shifting, all numbers shifted to right by k to their index increase by k. For the ones that $q_{old} + k$ is bigger than n, it goes to the front so the index mod by n. Since the index of median in the original array is $\frac{n}{2}$, the index of median in the shifted array is $A[(k + \frac{n}{2})\ mod\ n]$. □

Proof of Running time:

Proof. There are two procedures and Find Median is the main procedure. In the procedure Find K, I try to find the element k by binary search. For lines 14-21, the binary search in array A of n elements takes $O(\log n)$ time. The rest part in Find K takes O(1) time. So the running time for Find K is $O(\log n)$. For the procedure Find Median, line 2 use the procedure Find k,which takes $O(\log n)$ algorithm. For the lines 3-7, it is only a if which takes O(1) time. Therefore, the total time complexity of the algorithm is $O(\log n)$. As we know, $\log n = o(\sqrt{n})$. □

3 Problem #3: Network Flow (60 points)

You are given a flow network G, and G has edges entering the source s. Let f be an integer flow in G in which one of the edges (v, s) entering the source has $f(v, s) = 2$. Prove formally that there must exist another flow f' with $f'(v, s) = 0$ such that $|f| = |f'|$.

Hint: Maybe you can use a result from your homework?

Solution:

Proof. Wee try to prove this statement by transform flow f to flow f'.There must exist at least one cycle which contains the edge (v, s). Every vertex lies on some path starting from s. Since f satisfies conservation of flow, this cycle must exists. We can simply use a DFS to find such a cycle with no edges of zero flow in O(E). Then there are two cases:
1.Case 1: the minimum flow on this cycle is 2. Then we can simply decrement the flow of every edges in this cycle by 2. After this decrementing, we get the network with the same flow as $|f'|$. This decrement is valid because this preserves the value of flow so it is still the max flow. It won't violate the capacity constraint because before decrementing each flow on the cycle is at least 2, so the flow after decrementing must be bigger or equal to zero and less than capacity. Besides, flow conservation isn't violated because we decrement both incoming and outgoing edge for each vertex on the cycle by the same amount.
2.Case 2: the minimum flow on this cycle is 1. Then we can simply decrement the flow of every edges in this cycle by 1. After that, since $f(v, s) = 1$ now, we can still find another cycle which contains the edge (v, s). This cycle must exists since the new f still satisfies conservation of flow. We can simply use a DFS to find such a cycle with no edges of zero flow. Then we can simply decrement the flow of every edges in this cycle by 1.
These two decrementings are both valid because of the similar reason as above. These decrementing both preserve the value of flow so it is still the max flow. They won't violate the capacity constraint because before both decrementing each flow on the cycle is at least 1, so the flow after decrementing must be bigger or equal to zero and less than capacity. Besides, flow conservation isn't violated because we decrement both incoming and outgoing edge for each vertex on the cycle by the same amount in both decrementing.
Therefore, we have found out that each flow network with f(v,s) = 2 can be transformed to another flow with f'(v,s) = 0 with the way above. So there must exist

another flow f' with $f'(v, s) = 0$ such that $|f| = |f'|$. $\square$

4 Problem #4: Dynamic Programming (60 points)

You are given an $n \times n$ binary matrix (each entry of the matrix is either "0" or "1"), design a dynamic programming algorithm to find the largest rectangle containing only 1's and return its area, prove correctness and provide running time analysis. Full score will given for the solution with the running time $O(n^2)$.

For example, given the following matrix,

$$\begin{bmatrix} 1 & 0 & 1 & 0 & 0 \\ 1 & 0 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 \\ 1 & 0 & 0 & 1 & 0 \end{bmatrix}$$

the largest rectangle containing only 1's is the following and its area is 6.

$$\begin{bmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \end{bmatrix}$$

Solution: You can find the pseudocode of my algorithm below.

Proof of correctness: In this algorithm, our purpose it to find the largest rectangle of 1s, so my algorithm can be divided into two part. The first part is to use dynamic programming to find the size of longest column of continuous '1's vertically connected and end at mat[i][j]$(i, j \in [0, n])$. So for each row, the result of this dp is just like a Histogram with width of n and different heights. The second part is to use a stack to calculate the value of maximum area in the Histogram above, and this maximum area in the Histogram is the same as the largest rectangle in the original matrix.
At first we need to prove the correctness and time complexity of dynamic programming part.

1. Define the subproblems (and write down what it means). If not for anything but to help you define the algorithm.

    - DP[i][j] = The size of longest column of continuous '1's vertically connected and end at mat[i][j](the j_{th} element in the i_{th} row)(the bottom right corner is mat[i][j])

Algorithm 2 find largest rectangle

```
 1: procedure FIND LARGEST RECTANGLE($mat$)
 2:     $ans \leftarrow 0$
 3:     $dp[n][n] \leftarrow 0$ for all dp
 4:     for $i \leftarrow 0, n$ do
 5:         for $j \leftarrow 0, n$ do
 6:             if $mat[i][j] == 1$ then
 7:                 $dp[i][j] \leftarrow (i == 0)$ ? 1 : $dp[i-1][j] + 1$
 8:             else
 9:                 $dp[i][j] \leftarrow 0$;
10:             end if
11:         end for
12:         $ans \leftarrow max(ans, Find\ Largest\ Rectangle\ On\ One\ Row(dp[i]))$
13:     end for
14:     return ans
15: end procedure
16: procedure FIND LARGEST RECTANGLE ON ONE ROW($dp$)
17:     $stk \leftarrow Stack < int >$
18:     $stk.push(-1)$
19:     $maxvalue \leftarrow 0$
20:     for $i \leftarrow 0, n$ do
21:         while $stk.peek() \neq -1$ and $dp[stk.peek()] \geq dp[i]$ do
22:             $maxvalue \leftarrow max(ret, dp[stk.pop()] \times (i - stk.peek() - 1)$  ▷
    The difference between peek() and pop() is stk.peek() only returns the value
    on the top, stk.pop() not only return the top value but also pop it
23:         end while
24:         $stk.push(i)$
25:     end for
26:     while $stk.peek() \neq -1$ do
27:         $maxvalue \leftarrow max(maxvalue, dp[stk.pop()] * (n - stk.peek() - 1)$
28:     end while
29:     return maxvalue
30: end procedure
```

2. Write down a recurrence showing how the larger subproblem can be solved in terms of the smaller ones.

   - $$DP[i][j] = \begin{cases} 0 & mat[i][j] = 0 \\ 1 & mat[i][j] = 1 \; and \; i = 0 \\ DP[i-1][j] + 1 & mat[i][j] = 1 \; and \; i \neq 0 \end{cases}$$

3. Write down which subproblem gives the answer to the original question.

   - Output DP[i][j] for every i, j and send it to another procedure to get final answer.

4. Prove correctness by proving the "optimal substructure property". I.e. The recurrence relation defined by 2. is the solution for the subproblem as defined by 1.

   - We want to prove that DP[i][j] is the value of the size of longest column of continuous '1's vertically connected and end at mat[i][j](the j_{th} element in the i_{th} row)
   - To do so, we will induct on i.

     (a) Base Case 1(i=0), the recurrence relation is correct. Since there's no other row above, the DP value is only related to the current value in this line. If mat[i][j] = 0, then DP[i][j] = 1, else if mat[i][j] = 1, then DP[i][j] = 1. We can get this result from the first two cases in the recurrence. True.

     (b) IH : for i $\leq$ k, DP[i][j]($j \in [0, n]$) is the size of longest column of continuous '1's vertically connected and end at mat[i][j].

     (c) Induction Step: For i = k + 1 and j($j \in [0, n]$) there are two possible cases —

       - Case 1 : mat[i][j] is equal to 0. Obviously, when mat[i][j] = 0, there is no continuous '1's ending at mat[i][j], so the DP[i][j] = 0. True
       - Case 2 : mat[i][j] is 1. Since mat[i][j] is directly connected with mat[i-1][j], the size of longest column of continuous '1's vertically connected ending at mat[i][j] can be seen as the extension by 1 of the size of longest column of continuous '1's vertically connected ending at mat[i-1][j]. For the equation DP[i][j] = DP[i - 1][j] + 1, from out IH we know that DP[i-1][j] is the the size of longest column of continuous '1's vertically connected ending at mat[i-1][j]. So DP[i][j] is the size of

longest column of continuous '1's vertically connected ending at mat[i][j]. True.
  - Since there are only 2 cases, and they are both right, our claim is true.

5. Prove runtime by counting the number of subproblems.

   - For this DP algorithm, since we can use memoization to store the results of each sub problem, the runtime of the problem depends on the the number of times each sub problem gets called. As we can see from the algorithm lines 6-10, each iteration only calls 1 time of subproblems for calculating the DP value. Because there are $O(n^2)$ subproblmes $DP[i][j]](i, j \in [0, n]$ and we do O(1) in each subproblems, the runtime of the DP part is $O(n^2)$. The running time of the rest part is given in later proof.

Now we come to the second part. The purpose of this part is to get the maximum rectangle area in the histogram with heights equals to corresponding values of DP array.
Then we need to prove the second part that we, with stack, can get the maximum rectangle area in the histogram with heights equals to corresponding values of DP array. This way to use a stack is often called Monotone Stack.
In our algorithm in the procedure FIND LARGEST RECTANGLE ON ONE ROW, we try to calculate the maximum rectangle are of Histogram with height of DP. So we try to find the biggest value by examining all possible values. For each j here, we try to calculate the maximum possible area with the height equal to DP[i][j] and the width as large as possible. These include all possible values of biggest area because the biggest rectangle must extend itself as large as possible, which means that it's height must be equal to some height in the histogram and its width must be as large as possible. So we only need to calculate the maximum area with each height. And it is obvious that the biggest area of one height must be as wide as possible.
So for height dp[j], the left bar must be decided by the biggest value k1 that the height dp[k1] $<$ dp[j]. For similar reason, the right bar must be decided by the biggest value k2 that the height dp[k2] $<$ dp[j]. At some time, k1 and k2 is the left/right side of the graph. So now we need to find k1 and k2. In my algorithm, I use a stack to store an nondecreasing order of heights when I scan from left to right. If I find a new height which is lower than the last element of the stack ,I will pop the last element and let the new height to replace the old height as much as possible.

As we know, each number of height in DP can only get in and out the stack once. So by this way I can make sure that when I come to place j to examine the biggest rectangle with height dp[j], the first element that is smaller than dp[j] in the stack is the last element that is smaller than dp[j], which means that is the left bar of rectangle with height as dp[j]. And when an element in the stack is poped, it means that the number that poped it is the first on the right that its height is smaller than dp[j], which can be used as right bar. And if there's nothing in the stack, then the left bar is the left border of the histogram. For the numbers that are never poped, their right bar is the right border of this histogram. So by this way we find out the left and right bar of the rectangle with height DP[j] and we are able to calculate the area of this rectangle. After comparing areas of all possible rectangles, we can find the maximum rectangle area.

And we also need to prove that the maximum rectangle area in histogram of dp is the largest rectangle in the matrix.
This is very obvious since maximum rectangle area in Histogram of DP is just the same shape of rectangle of '1's in the original matrix. Since for each row, we can find the maximum rectangle area in Histogram of DP, which corresponds to a largest rectangle ending at point(i,j)(the bottom-right corner is (i,j)). Then we can say that for each point (i,j) in the matrix., we have searched largest possible rectangle ending at it. So we have found all possible answers in the matrix, and the result of our algorithm must be the largest rectangle of '1's.
Therefore, this algorithm can find the maximum rectangle in the matrix.

Proof of running time:

Proof. As we have proved in the correctness part of the dynamic programming, we know that the overall running of dynamic programming takes $O(n^2)$ time.
And for the procedure FIND LARGEST RECTANGLE ON ON, every time we call this procedure, it takes $O(n)$ time. This is because the for loop in lines 20-25 except the while loop takes $O(n)$ time since it goes through the n element in the dp array and push them into stk. And In this loop, the while also takes only $O(n)$ time in each call of this procedure because each iteration of while loop will pop an element from stk, and there are at most n elements in stk. For lines 26-28, the while loop only takes at most $O(n)$ time because each iteration of while loop will pop an element from stk, and there are at most n elements in stk. Therefore, the running time of each call of procedure is $O(n) + O(n) + O(n) = O(n)$.

In the main procedure FIND LARGEST RECTAGLE, we call the second procedure only n times, so the overall running time of this stack procedure is $O(n\, timesn) = O(n^2)$.
Since these are two main functions of this algorithm and the other lines are trivia with little running time, the overall running time of the whole algorithm is $O(n^2)$
$\square$

5 Problem #5: MST (60 points)

Let $G = (V, E)$ be an undirected, connected and weighted graph with the weight function w. Assume that there exist an edge e_0 and two numbers X, Y such that $X < Y$, $w(e_0) = Y$ and $w(e) = X$ for all $e \in E, e \neq e_0$. Design an efficient algorithm that finds an MST in G. Prove the correctness of your algorithm and give the running time. Your answers must be explained clearly, and with enough details. The full credit will be given if the running time of your algorithm is $o(|E| \log(|V|))$.

Solution:

Algorithm 3 MST

```
1: procedure KRUSKAL WITHOUT SORTING($G$)
2:     $A \leftarrow \emptyset$
3:     for each vertex $v \in G.v$ do
4:         $Make - Set(v)$        ▷ Use the Disjoint-Set taught in class as Kruskal
5:     end for
6:     Scan all edges and find the edge $e_0$ with largest weight
7:     Set the edges of G.E into nondecreasing order by weight without sort, but
   by putting all edges other than $e_0$ in any order before $e_0$ and setting $e_0$ as the
   last edge.
8:     for each edge $(u, v) \in G.E$, taken in nondecreasing order by weight do
9:         if $Find - Set(u) \neq Find - Set(v)$ then
10:            $A = A \cup \{(u, v)\}$
11:            $Union(u, v)$
12:        end if
13:    end for
14: end procedure
```

Proof of correctness:

Proof. This algorithm is very similar to the Kruskal Algorithm we taught in class. The only difference is how to set all edges in nondecreasing order. Since we have proved the correctness of Kruskal Algorithm, we only need to prove that the setting order parts can get the same results with the sorting part in the original Kruskal. It is obvious that since all weights of edges in G.E is X except the weight of e_0 is Y,and Y is bigger than X, so setting all other edges before e_0 will get a nondecreasing order as we want. Since this part serves as the same function of the part in the sorting part of Kruskal, and the rest part of this algorithm is the same as Kruskal, we can say that this algorithm is just doing the same thing as the Kruskal algorithm with different time complexity. Therefore this algorithm can get the MST as Kruskal can get MST. The correctness is proved □

Proof of Running time:

Proof. As we know from Chapter 21.4 in CLRS, A sequence of m MAKE-SET, UNION, and FIND-SET operations, n of which are MAKE-SET operations, can be performed on a disjoint-set forest with union by rank and path compression in worst-case time $O(n\alpha(n))$. $\alpha(n)$ is a very small number for very large $|V|$ and is less than log(V) as mentioned in CLRS. Besides, since we can find a MST requires the number of edges $|E|$ is at least $|V| - 1$, so we have $|V| = O(|E|)$. In this algorithm, lines 2 - 5 are $O((|V| + |E|)\alpha(|V|) = O(|E|\alpha(|V|))$ with only a loop and make-set. For lines 6-7, we only scan through all edges and find the edge with largest weight and put it in the back, so the time complexity is $O(|E|)$. For lines 8 - 13, it's a loop of all edges and each operation in it is $O(\alpha(|V|))$, so the total time complexity of these lines are $O(|E|\alpha(|V|))$. In summary, the total complexity is $O(|E|\alpha(|V|) + O(|E|) + O(|E|\alpha(|V|)) = O(|E|\alpha(|V|))$, and we know that $O(|E|\alpha(|V|)) = o(|E|\log(|V|))$ □

6 Problem #6: Median (60 points)

For n distinct numbers $x_1, x_2, \ldots, x_n$ with distinct positive weights $w_1, w_2, \ldots, w_n$ such that $\sum_{i=1}^{n} w_i = 1$,

the **weighted (lower) median** is the number x_k satisfying

$$\sum_{x_i<x_k} w_i < \frac{1}{2}$$

and

$$\sum_{x_i>x_k} w_i \leq \frac{1}{2}$$

For example, if numbers are $0.1, 0.35, 0.05, 0.1, 0.15, 0.05, 0.2$ and each number equals its weight (that is $w_i = x_i$ for $i = 1, 2, \ldots, 7$), then the median is 0.1, but the weighted median is 0.2.

Devise an algorithm that computes the weighted median in $\Theta(n)$ worst-case time. Prove correctness and provide running time analysis. 30 points will be given for the algorithm with running time $O(n \log n)$.

Solution: My algorithm uses the SELECT algorithm with worst case O(n) time complexity in Chapter $9,3$ in CLRS. This algorithm is very similar to the Randomized-SELECT algorithm. It goes through the following steps:
Step 1. Using the SELECT algorithm in Chapter 9.3 in CLRS to find the median x_{mid} of the array
Step 2. Partition the original array into 2 parts around the median using the PARTITION algorithm from QuickSort mentioned in CLRS Chapter 7. The result is that all 'x's whose $x_i < x_{mid}$ is in the left part and all 'x's whose $x_i > x_{mid}$ is in the right part
Step 3.x_{mid} is the position of the median we found in the previous step. Calculate $W_L = \sum_{x_i<x_{mid}} w_i$ and $W_R = \sum_{x_i>x_{mid}} w_i$. Here are three possible cases:
(1) $W_L < \frac{1}{2}$ and $W_R \leq \frac{1}{2}$. In this case, x_{mid} is the weighted median we want.
(2) $W_L \geq \frac{1}{2}$. In this case, the weighted median is before x_{mid}. We can set $w_{mid} = w_{mid} + W_R$ and repeat to step 1 on the left half of the array(including x_{mid}).
(3) $W_R \geq \frac{1}{2}$.In this case,the weighted median is after x_{mid}. We can set $w_{mid} =$

$w_{mid} + W_L$ and repeat to step 1 on the right half of the array(including x_{mid}). After all these steps and loops, we will find the weighted median in step 3 case 1.

Proof of correctness:

Proof. In step 1 we have found the median in the array. And in step 2 we have partition the array into two parts around the median x_{mid}. The correctness of these 2 steps are mentioned on CLRS.
So let's see what we have after these 2 steps. The array after loop 2 is an array of x_{mid} in the middle. On the left is all 'x's that are smaller or equal to x_{mid}. On the right are 'x's that are bigger or equal than x_{mid}. Therefore, the sum of the weights of left part is $W_L = \sum_{x_i < x_k} w_i$, and the sum of weights of right part is $W_R = \sum_{x_i \geq x_k} w_i$. These two values are the standard of whether x_{mid} is the weighted median. Since we have divided the array int two part that will never cross again when calculating sum, if the sum of weight of one part is bigger than $\frac{1}{2}$, it means that the median weight will never be in the other part. Therefore, we should go search for the weighted part in the part with weight bigger than $\frac{1}{2}$. If for current $W_L < \frac{1}{2}$ and $W_R \leq \frac{1}{2}$, just as case 1 in step 3, then it is the weighted median according to definition and we have found the answer. Otherwise, we should look for weighted median in the part with sum of weight bigger than $\frac{1}{2}$, just like what we do in case 2 and case 3 in step 3. Under this case, to make sure that the weighted median is the same as the previous array, we can add the sum of weight to w_{mid}. It is valid because that the weighted median must be in the side of larger sum of weight, so there's no difference for calculating the weight of the smaller half on one element or on many element. This keeps the total sum of weight equals to 1 without and effect of the weighted median. We can see that the searching area becomes smaller and smaller for each iteration because we halve the searching area each time ins step 3. So this loop must has an end with case1 in step 3. In the worst case we will get to a state that only 1 number needs to be searched the its weight is 1, and that is the answer. Since the loop must end with. Therefore, our algorithm can get the weighted median.
For the sufficiency of proof, now we gonna prove Step3 by induction:
1.Base Case: There is only one element in the array. Then we will return that element since the weight of it must be 1.
2.IH: For all array with k or less elements, our algorithm can get the weighted median.
3.IS: Now we have an array with k + 1 elements. And we go through these steps on this array.
As I mentioned above for step 1 and step 2, now we get a array with $x_i \leq x_{mid}$ on

the left and $x_i \geq x_{mid}$ on the right.
Then for step 3, We need to prove that the weighted median of the array after step 3 is the weighted median of the original array.
Case 1: $W_L < \frac{1}{2}$ and $W_R \leq \frac{1}{2}$. It is obvious that the median is the weighted median according to definition. True.
Case 2: $W_L \geq \frac{1}{2}$. In this case, since W_L is bigger than $\frac{1}{2}$, the weighted median must be on this side. So we can only look for weighted median on the left part. Under this case, to make sure that the weighted median is the same as the previous array, we can add the sum of weight to w_{mid}. It is valid because that the weighted median must be in the side of larger sum of weight, so there's no difference for calculating the weight of the smaller half on one element or on many element. This keeps the total sum of weight equals to 1 without and effect of the weighted median. So the problem with k+1 elements becomes a problem with $\frac{k+1}{2} + 1$ elements. According to IH, this is true.
Case 3: $W_R \geq \frac{1}{2}$ In this case, since W_R is bigger than $\frac{1}{2}$, the weighted median must be on this side. So we can only look for weighted median on the left part. Under this case, to make sure that the weighted median is the same as the previous array, we can add the sum of weight to w_{mid}. It is valid because that the weighted median must be in the side of larger sum of weight, so there's no difference for calculating the weight of the smaller half on one element or on many element. This keeps the total sum of weight equals to 1 without and effect of the weighted median. So the problem with k+1 elements becomes a problem with $\frac{k+1}{2} + 1$ elements. According to IH, this is true.
In summary, The step 3 is true. The whole algorithm is true. □

Proof of Running time:

Proof. All our analysis below is based on the worst case. In step 1, we use the SELECT algorithm in Chapter 9.3 whose worst case time complexity is $O(n)$. In Step 2, we also use the partition algorithm whose worst time complexity is also $O(n)$ according to Chapter 7 in CLRS. And since in each iteration in step 1 we find the median of the array, and the step 3 we only enter a new iteration with array of half size + 1, we can say that in our algorithm, the equation $T(n) = T(\frac{n}{2} + 1) + \Theta(n)$ is satisfied in this algorithm, in which T(n) stands for the time complexity of our algorithm. And we know that $T(1) = 1$. Therefore, $T(n) = T(\frac{n}{2} + 1) + \Theta(n) = ... = \sum_{i=0}^{\log n}(\frac{n}{2^i} + 1) + \Theta(n) + \Theta(\frac{n}{2}) + ... + \Theta(1) = n\sum_{i=0}^{\log n}\frac{1}{2^i} + \sum_{i=0}^{\log n} 1 + \Theta(n) \leq 2n + \log n + 1 + \Theta(n) = \Theta(n)$ □