Worth: 12% **Due:** By 5:59pm on Tuesday 10 February

Remember to write the *full name* and *student number* of *every group member* prominently on your submission.

Please read and understand the policy on Collaboration given on the Course Information Sheet. Then, to protect yourself, list on the front of your submission ***every*** *source of information you used to complete this homework (other than your own lecture and tutorial notes). For example, indicate clearly the* ***name*** *of every student from another group with whom you had discussions, the* ***title and sections*** *of every textbook you consulted (including the course textbook), the* ***source*** *of every web document you used (including documents from the course webpage), etc.*

For each question, please write up detailed answers carefully. Make sure that you use notation and terminology correctly, and that you explain and justify what you are doing. Marks ***will*** *be deducted for incorrect or ambiguous use of notation and terminology, and for making incorrect, unjustified, ambiguous, or vague claims in your solutions.*

---

1. Considering the follow algorithm which searches for the last appearance of value k in an array A of length n. The index of the array starts from 0.

```
FindLast($A, k$):
1   for $i \leftarrow n-1, n-2, \ldots, 0$:
2       if $A[i] = k$:
3           return $i$
4   return $-1$
```

The input array A is generated in the following specific way: for $A[0]$ we pick an integer from $\{0, 1\}$ uniformly at random; for $A[1]$ we pick an integer from $\{0, 1, 2\}$ uniformly at random; for $A[2]$ we pick an integer from $\{0, 1, 2, 3\}$ uniformly at random, etc. That is, for $A[i]$ we pick an integer from $\{0, \ldots, i+1\}$ uniformly at random. All choices are independent from each other. Now, let's analyse the complexity of FindLast by answering the following questions. All your answers should be in **exact form**, i.e., **not** in asymptotic notations.

NOTE: For simplicity, we assume that k is an integer whose values satisfies $1 \leqslant k \leqslant n$.

(a) In the **best case**, for input (A, k), how many times is Line #2 ("**if** $A[i] = k$") executed? Justify your answer.

(b) What is the probability that the best case occurs? Justify carefully: show your work and explain your calculation.

(c) In the **worst case**, for input (A, k), how many times is Line #2 executed? Justify your answer.

(d) What is the probability that the worst case occurs? Justify carefully: show your work and explain your calculation.

(e) In the **average case**, for input (A, k), how many times is Line #2 expected to be executed? Justify your answer carefully: show your work and explain your calculation.

HINT: Your answers should be in terms of both n and k. Thinking about the number of comparisons needed to find a given k, which values are possible, which values are not?

2. This question consists of two problems that are seemingly unrelated to each other, but we will be able to solve both of them using a similar approach based on a data structure that we learned in class.

   (a) Suppose we have m sorted arrays (all in non-descending order) with a total number of n elements, and we would like to merge these arrays into one big sorted array. Devise an algorithm with worst-case runtime $\mathcal{O}(n \log m)$, using a data structure that we learned. Your data structure can hold **at most m** elements. Give a detailed description of your algorithm (write pseudo-code if necessary), and argue that your algorithm works correctly and has the desired worst-case runtime.

   HINT: Start by thinking about how to merge two sorted arrays. Do **not** use an AVL tree (or other balanced binary search tree)!

   (b) Let a, b, c, d be four integers whose values are between 1 and n, inclusive. Devise an algorithm that finds all solutions to the following equation

   $$a^5 + b^5 = c^5 + d^5$$

   Your algorithm should have worst-case runtime $\mathcal{O}(n^2 \log n)$. You should use a heap of size **at most n**. Give a detailed description of your algorithm, and argue why your algorithm works correctly and has the desired worst-case runtime.

   HINT: How is this algorithm related to the one in the previous part? What corresponds to the m sorted arrays? How can we obtain the solutions to the equation from the merged sorted array?

3. Consider the following abstract data type that we will call a "PIXELSET."

   Objects: A set S of "coloured pixels" that are represented by triples (x, y, c), where x and y are positive integers denoting a position of a pixel on the screen, and $c \in \{R, B, G\}$ denotes a colour. Note that each position can be assigned up to three colours, e.g., $(5,3,R)$, $(5,3,B)$ and $(5,3,G)$ can co-exist in a PIXELSET, but there cannot be more than one "$(5,3,R)$".

   Operations:

   - READCOLOUR(S, x, y): Return the colours at position (x, y), i.e., the set $\{\, c \mid (x, y, c) \in S \,\}$.
   - WRITECOLOUR(S, x, y, c): Assign colour c to position (x, y), i.e., add the triple (x, y, c) to S. If position (x, y) already has colour c, then do nothing.
   - NEXTINROW(S, x, y): Return the position of the next coloured pixel that appears after (x, y) and in the same row as (x, y), i.e., return $(x, \min\{\, y' \mid y' > y \text{ and } (x, y', c) \in S \text{ for some } c \,\})$. Return $(0, 0)$ if no such coloured pixel exists. **Assumption on input values:** You can assume that a pixel at (x, y) **exists** in the PIXELSET.
   - NEXTINCOLUMN(S, x, y): Similar to NEXTINROW, return the position of the next coloured pixel that appears after (x, y) and in the same column as (x, y). **Assumption on input values:** You can assume that a pixel at (x, y) **exists** in the PIXELSET.
   - ROWEMPTY(S, x): Return whether Row x is empty, i.e., return TRUE if and only if there does not exist a triple (x, y, C) with the given x in S.
   - COLUMNEMPTY(S, y): Similar to ROWEMPTY, return whether Column y is empty.

   Requirements: All above operations must have worst-case runtime $\mathcal{O}(\log n)$, where n is the total number of coloured pixels in the PIXELSET S.

   Give a *detailed* description of how to use AVL trees to implement PIXELSETS. In particular, answer the following questions.

(a) How many AVL trees are you using? What does each node correspond to? What information is stored in each node?

(b) What are the keys that you use for sorting each of the AVL trees? For each AVL tree, define **carefully and precisely** how you compare two pixels positioned at (x, y) and (x', y').

(c) For each of the above operations, describe in detail how it works, and argue why it works correctly and why its worst-case runtime is $\mathcal{O}(\log n)$.

HINT: Try to make use of textbook algorithms for BST and AVL trees, and please do **not** repeat algorithms or runtime analyses from class or the textbook—just refer to known results as needed.

4. Suppose you are working for a tech company named QUACKER, which provides a novel web service where users can post short messages (*a.k.a.* "quacks") which usually contain *hashtags*. Your job, as a *Senior Hashtag Master*, is to design data structures that support real-time hashtag-related statistics, i.e., the data structures are being dynamically populated and modified in real-time, and the statistical results you get from the data structures must be up-to-date at any time. Below is the detailed description of the ADT, named HASHQUACKSTORE.

Objects: A multiset H of hashtags, which are simply strings such as "`#myweirdrelative`" and "`#uoft`", assuming all letters are in lowercase. Each hashtag has a *count* which indicates the total number of times this hashtag occurs in the multiset H. (A *multiset* is like a set where repeated elements make a difference. For example, $\{a, b, a, c\}$ and $\{c, a, c\}$ represent **different** multisets.)

Operations:

- GETCOUNT(H, s): Return the count of hashtag s in H.
- TOPTHREE(H): Return the top 3 most used hashtags in H. In case there is a tie at the third position, choose one arbitrarily. In case there are fewer than 3 different hashtags, return all of them. In other words, this operation always returns at most 3 hashtags.
- ADD(H, s): Add a new use of hashtag s to H, i.e., the count of s stored in H should be incremented properly. If s does not exist in H yet, set its count to 1.
- COUNTBETWEEN(H, a, b): Return the total **number** of unique hashtags in H whose counts are between a and b, *inclusive*. Assume that a and b are positive integers, and that $a \leqslant b$.

Requirements: Let n be the total number of unique hashtags stored in the ADT.

- GETCOUNT has average case runtime $\mathcal{O}(1)$.
- TOPTHREE has worst-case runtime $\mathcal{O}(1)$.
- ADD has average-case runtime $\mathcal{O}(\log n)$.
- COUNTBETWEEN has worst-case runtime $\mathcal{O}(\log n)$.
- The space complexity (i.e., the total number of elements stored in a HASHQUACKSTORE) is in $\mathcal{O}(n)$.

Describe your design of HASHQUACKSTORE by answering the following questions.

(a) Which data structures are you using in your design? What is the purpose of each of the data structures you use?

(b) What information do you store in each element of each of your data structures? Why can the information be maintained efficiently when the data structure is being modified?

(c) How does GETCOUNT(H) work? Why is its average-case runtime $\mathcal{O}(1)$? HINT: For the average-case runtime, you don't need to perform the detailed analysis like what you did in Question 1. Just use analysis results from lectures, tutorials or textbook. This hint also applies to Part (e).

(d) How does TOPTHREE(H) work? In particular, how do you determine the top first, second and third hashtags? Why is it $\mathcal{O}(1)$ in worst-case?

(e) How does ADD(H, s) work? How do you make sure *all* data structures are correctly updated? Why is it $\mathcal{O}(\log n)$ in average-case?

(f) How does COUNTBETWEEN(H, a, b) work? In particular, how do you calculate the return value? Why does it run in $\mathcal{O}(\log n)$? Beware that there could be multiple hashtags with the same count, please explain why your algorithm works correctly when this is the case. **You may assume** that, in the data structure, there already exist hashtags whose counts are a and b.

(g) **BONUS.** (*This part is worth a very small amount of additional marks to this question. Only work on this if you have finished other parts.*) In Part (f), if we remove the assumption that there already exist hashtags with counts a and b, how do you modify your COUNTBETWEEN(H, a, b) operation so that it still works correctly?

HINT: Consider using different data structures to meet different requirements. Consider augmenting some of the data structures when necessary.

1. (a) In the best case, $A[n-1]$, the first element we check, is k, so we find k after one comparison, i.e., Line #2 is executed only once.

   (b) The probability of the best case is the probability that we choose k for $A[n-1]$, out of $n+1$ candidates ($\{0, 1, \ldots, n\}$). Since it's chosen uniformly at random, this probability is $1/(n+1)$.

   (c) In the worst case, k does not exist in A, and the algorithm will go through the whole array, i.e., executing Line #2 for n times.

   (d) The probability of the worst case is the probability that k is not chosen by any of the entries $A[0]$, $A[1]$, …, $A[n-1]$. For each one entry, the probability of not choosing k is different.

   For entries before $A[k-2]$ (including $A[k-2]$), the probability of not choosing k is simply 1, because k is not even a candidate.

   For entries after $A[k-1]$ (including $A[k-1]$), k becomes a candidate. For $A[k-1]$, the probability of not choosing k is $k/(k+1)$ (choose anything but k out of $k+1$ candidates). Similarly, for $A[k]$, the probability of not choosing k is $(k+1)/(k+2)$. In general, for $A[i]$ ($k-1 \leqslant i \leqslant n-1$), the probability of not choosing k is $(i+1)/(i+2)$.

   Since all choices are made independently, we just take the product of probabilities of k not chosen by each entry to get the probability that k is not chosen by any entry, i.e.,

$$\Pr(k \text{ not in } A) = \prod_{i=0}^{n-1} \Pr(k \text{ not chosen by } A[i]) = \prod_{i=k-1}^{n-1} \frac{i+1}{i+2} = \frac{k}{n+1}$$

   However, there is a special case when $k = 1$, because the worst-case (n comparisons) happens in two possible ways. One is that k does not exist in A; the other is that k is picked by $A[0]$ but we do n comparisons anyway. The probability of the former way is $1/(n+1)$ by plugging in $k = 1$ to the above equation; the probability of the latter way is also $1/(n+1)$ because $A[0]$ chooses 0 and 1 with equal probabilities. So overall the probability of worst-case when $k = 1$ is $2/(n+1)$.

   The final answer to the worst-case probability is

$$\Pr(\text{worst-case}) = \begin{cases} k/(n+1) & \text{if } 2 \leqslant k \leqslant n \\ 2/(n+1) & \text{if } k = 1 \end{cases}$$

   (e) For the average case, we need to determine the probability that we need to perform t ($1 \leqslant t \leqslant n$) comparisons until finding (or not finding) k. Let t_n be a random variable that denotes the number of comparisons (Line #2) executed. The first thing to note is that only certain values are possible for t_n, which are $1, 2, \ldots, n-k+1$ and n. That is, either you find k within $n-k+1$ steps, or you never find it (after $n-k+1$ steps, k would not be a candidate any more).

   Now let's figure out the probability of $t_n = 1, 2, \ldots, n-k+1$, when k is found. The probability that $t_n = t$ is the probability that k is not chosen by $A[n-1]$, $A[n-2]$, …, $A[n-t+1]$ and is chosen by $A[n-t]$, i.e.,

$$\Pr(t_n = t) = \frac{1}{n-t+2} \cdot \prod_{i=n-t+1}^{n-1} \frac{i+1}{i+2} = \frac{1}{n+1}$$

   Note that, this probability is independent of t as long as we are given that k is found. Then the probability that k is not found is simply 1 minus the sum of probabilities of all k-found cases (also the result of Part (d)), which is

$$\Pr(k \text{ not found}) = 1 - (n-k+1) \cdot \frac{1}{n+1} = \frac{k}{n+1}$$

Thus the complete description of t_n's probability distribution is

$$\Pr(t_n = t) = \begin{cases} 1/(n+1) & 1 \leqslant t \leqslant n-k+1 \ \ (k \text{ found}) \\ k/(n+1) & t = n \ \ (k \text{ not found}) \\ 0 & \text{otherwise} \end{cases}$$

Note that when $k = 1$, t_n could be n for two reasons: found at the last comparison or not found. Finally, the average running time of FINDLAST, with the given input distributions, is

$$E[t_n] = n \cdot \frac{k}{n+1} + \sum_{i=1}^{n-k+1} i \cdot \frac{1}{n+1} = \frac{2nk + (n-k+2)(n-k+1)}{2(n+1)}$$

2. (a) When we merge two arrays into one sorted array, we use two pointers, initially pointing at the beginning of each array. Then we compare the two values pointed by the two pointers and increment the one that is pointing to the smaller value (after appending to smaller value to the output array). We keep incrementing the pointers until one of them reaches the end of its array. Then, append to the output whatever is left in the other array, and we will get a merged sorted array. The running time of this algorithm is $\mathcal{O}(n)$ since we visit each element at most once. The thing to note here is that, at each iteration, we needed to determine the minimum of *two* values being pointed to, which takes constant time.

   Merging m sorted arrays is essentially the same as merging two sorted arrays, except that at each iteration we need to determine the minimum of m values being pointed to, instead of 2. If we find the min by just going through the m values linearly, that will result in a $\mathcal{O}(nm)$ overall running time. We can do it more efficiently using a min-heap. We maintain a heap with m elements where each element corresponds to a pointer, and the key is the value that the pointer points to. At each iteration, we do the following:

   i. Get *minptr* by calling EXTRACTMIN on the heap.
   ii. Append the value pointed by *minptr* to the output array.
   iii. Increment *minptr*.
   iv. Insert *minptr* to the heap, with the new pointed value being its key.

   Since EXTRACTMIN and INSERT both take $\mathcal{O}(\log m)$ time, the overall running time of the merging is then $\mathcal{O}(n \log m)$.

   (b) This problem can be solved by performing a merge of n wisely-defined arrays. Let $R_{x,y}$ denote the triple $(x, y, x^5 + y^5)$. We have in total n^2 triples from $R_{1,1}$ up to $R_{n,n}$.

   Now we define the n sorted input arrays, namely, A_1 to A_n. Let A_i contain $R_{i,1}, R_{i,2}, \ldots, R_{i,n}$, i.e., A_1 contains $R_{1,1}, R_{1,2}, \ldots, R_{1,n}$, A_2 contains $R_{2,1}, R_{2,2}, \ldots, R_{2,n}$, etc. Based on this definition, the triples $(x, y, x^5 + y^5)$ in each array have the same x, have y in increasing order, and also have $x^5 + y^5$ in increasing order.

   Then we merge the n sorted arrays of triples $A_1, \ldots, A_n$, using the algorithm described in Part (a). We use the value of $x^5 + y^5$ as the comparison key while merging. As a result, the output will be a single array of triples $(x, y, x^5 + y^5)$ sorted according to $x^5 + y^5$. If the equation $a^5 + b^5 = c^5 + d^5$ has any solution, we would have **adjacent** triples in the output array that have the same $x^5 + y^5$, and the x's and y's in these triples are exactly the solutions a, b, c, d that we are looking for. That is, by going through the output array once, we can find all solutions to the equation. The worst-case running time of the merging, according to the analysis in Part (a), is $\mathcal{O}(n^2 \log n)$ since there are n arrays and n^2 elements in total.

   NOTE: Google "taxicab number" for more insights into this problem.

3. (a) We use two AVL trees with keys sorted in different ways. Each node corresponds to one position (x, y), and each nodes stores the set of colours (e.g., $\{R, G\}$) that are assigned to this position.

   (b) The keys for both trees are the position pairs (x, y). One AVL tree is row-first ordered ($\leqslant_R$), and the other is column-first ordered ($\leqslant_C$). In the row-first ordered tree, two keys are compared as follows:

   $$(x,y) \leqslant_R (x',y') \quad \text{iff} \quad x < x' \text{ or } ((x = x') \text{ and } (y \leqslant y'))$$

   This way of ordering makes sure that when the pixels are sorted, they are ordered in such a way that all rows are sorted from low to high, and within each row the pixels are sorted by their column number from low to high.

   Symmetrically, in the column-first ordered tree, two keys are compared as follows:

   $$(x,y) \leqslant_C (x',y') \quad \text{iff} \quad y < y' \text{ or } ((y = y') \text{ and } (x \leqslant x'))$$

   (c) Below is the description of how each operation works.

   - READCOLOUR(S, x, y): Perform a AVLSEARCH for key (x, y) in either of the AVL trees and return the colour set stored in the found node. If the node is not found, return an empty set. AVLSEARCH takes $\mathcal{O}(\log n)$ time.
   - WRITECOLOUR(S, x, y, c): Perform a AVLSEARCH for (x, y) first. If the node is found, then add colour c to the colour set stored in the found node (if c is not in the set yet). If the node is not found, then call AVLINSERT to insert a new node with key (x, y) and with $\{c\}$ stored as its colour set. This needs to be done to **both** AVL trees. This operation takes $\mathcal{O}(\log n)$ time because because both AVLSEARCH and AVLINSERT take $\mathcal{O}(\log n)$ time.
   - NEXTINROW(x, y): First, in the row-first ordered tree, use AVLSEARCH to locate the node p with key (x, y) (note that we assumed that the key must exist in the tree). Then call TREESUCCESSOR to obtained the successor of p with key (x', y'). If the successor does not exist, return $(0, 0)$; if $x' > x$, return $(0, 0)$; otherwise return (x', y'). Because of how row-first order ($\leqslant_R$) is defined in Part (b), this returned successor is exactly the next pixel after (x, y) in the same row. The running time is $\mathcal{O}(\log n)$ because both AVLSEARCH and TREESUCCESSOR take $\mathcal{O}(\log n)$ time in an AVL tree.
   - NEXTINCOLUMN(x, y): Exactly the same as NEXTINROW except that we do it in the column-first ordered AVL tree.
   - ROWEMPTY(x): First, insert a node with key $(x, 0)$ to the row-first ordered AVL tree, then call NEXTINROW$(x, 0)$, and return **True** if and only if NEXTINROW$(x, 0)$ returns $(0, 0)$, i.e., Row x is empty if and only if there is no next pixel in Row x after $(x, 0)$. Then don't forget to delete node $(x, 0)$ from the row-first ordered AVL tree. This operation runs in $\mathcal{O}(\log n)$ time because INSERT, DELETE and NEXTINROW are all in $\mathcal{O}(\log n)$. We can also avoid the INSERT and DELETE by carefully doing a TREESEARCH for $(x, 0)$, and choose the appropriate node to run NEXTINROW on.
   - COLUMNEMPTY(y): Exactly the same idea as ROWEMPTY except that we do it in the column-first ordered AVL tree. Insert $(0, y)$, check NEXTINCOLUMN$(0, y) = (0, 0)$, then delete $(0, y)$.

4. (a) To implement a HASHQUACKSTORE that satisfies all the requirements, we need to use a hash table, a max-heap and an augmented AVL tree.

   - We need a hash table to satisfy the requirement that GETCOUNT must be in $\mathcal{O}(1)$ in average-case since hash table supports constant-time lookup in average-case. The number of buckets in the hash table needs to be chosen in a sensible way such that the load factor α is reasonable.

- We need a max-heap to be able to perform TopThree in constant time. Also, insert and delete in a heap both take $\mathcal{O}(\log n)$ time, which also satisfies the requirements.
- We need an augmented AVL tree to be able to perform the CountBetween operation in $\mathcal{O}(\log n)$ time. Also, insert and delete in an AVL-tree both take $\mathcal{O}(\log n)$ time, which also satisfies the requirements.

(b) We store the following information in each of the data structures we use.

- In the hash table each node corresponds to a hashtag (string hashed into different buckets). Each node stores the hashtag string, the count of the hashtag, as well as a pointer to the hashtag's position in the max-heap.
- In the max-heap, each node corresponds to a hashtag. Each node store the hashtag string and its count. The heap is ordered according to the count of each node, i.e., the key is the count of each hashtag.
- In the augmented AVL tree, each node x corresponds to a count number which is also used as the key, and stores the number of hashtags ($x.num\_tags$) that have this count number, i.e., the node with key 1000 also stores how many hashtags have been used for 1000 times. In addition, as the augmentation, each node x stores the sum of the *num_tags* of all nodes in the subtree rooted at x, namely $x.sum$. We have that

$$x.sum = x.left.sum + x.right.sum + x.num\_tags$$

Therefore, $x.sum$ depends only on the information in nodes x, $x.left$ and $x.right$. According to Theorem 14.1, this attribute can be maintained efficiently during insertion and deletion without affecting their $O(\log n)$ running time.

(c) GetCount(s), just do a lookup in the hash table, which takes $\mathcal{O}(1)$ time in average-case.

(d) TopThree(): Use the heap H to get the top 3 used hashtags. $H[1]$ is clearly the first one, the larger one between $H[2]$ and $H[3]$ is the second largest one. The third one is slightly trickier: if the second largest one is $H[2]$ then the third largest one is the largest among $H[3]$ and the two children of $H[2]$, i.e., $H[4]$ and $H[5]$. Similarly, if the second largest one is $H[3]$ then the third largest one is the largest one among $H[2]$, $H[6]$ and $H[7]$. Since these are a constant number of comparisons, this operation runs in $\mathcal{O}(1)$ time in worst-case.

(e) Add(s): We first do a lookup in the hash table to determine if s is a new hashtag that has never appeared before. If s is new hashtag, first insert a new node to the hash table ($\mathcal{O}(1)$), then insert a new node to the heap with count 1 ($\mathcal{O}(\log n)$), update the node's index in the heap to the hash table ($\mathcal{O}(1)$), and then in the AVL tree, find the node with key 1 and increment its *num_tags* ($\mathcal{O}(\log n)$).

If s is an existing hashtag, we do the following. First, increment the count value of s in the hash table. Then call IncreaseKey by 1 on the node of s in the heap (note that all nodes involved in this operation need to have their indices updated in their corresponding hash table entry). Then we find the old count in the AVL tree, decrease its *num_tags* by 1 (delete the node if *num_tags* becomes zero), and find the node of the new count (incremented) and increase its *num_tags* by 1. If the new count does not exist, insert a new node with $num\_tags = 1$. All above operations are in $\mathcal{O}(\log n)$, so this operations takes $\mathcal{O}(\log n)$ time. This is average-case runtime since the lookup in the hash table is in average-case.

(f) COUNTBETWEEN(a, b): First, we need an operation that returns the total number of hashtags that have counts no larger than a given count k, this operation is exactly the PARTIALSUM operation that we designed in Problem Set 4, because we are basically computing the following partial sum

$$\text{PARTIALSUM}(k) = \sum_{x:\, x.key \leqslant k} x.num\_tags$$

Then the return value of COUNTBETWEEN(a, b) is

$$\text{COUNTBETWEEN}(a, b) \leftarrow \text{PARTIALSUM}(b) - \text{PARTIALSUM}(a) + x_a.num\_tags$$

where x_a is the nodes in the AVL tree that have key a, which can be found using TREESEARCH in $\mathcal{O}(\log n)$ time. Since PARTIALSUM takes $\mathcal{O}(\log n)$ time, COUNTBETWEEN also takes $\mathcal{O}(\log n)$ time.

Duplicates are handled correctly because we make sure all nodes in the AVL tree have distinct keys and we store the number of duplicates in each node. Allowing duplicate keys in an AVL tree would not work, since the rotations would break the order of the keys.

(g) If keys a and b may not exist in the AVL tree, we need to modify the TREESEARCH(k) operation such that it returns the node with largest key that is no larger than k, along with a flag indicating whether k is found (can be done in $\mathcal{O}(\log n)$ time). If the returned flag of TREESEARCH(a) is "not found," the equation for COUNTBETWEEN becomes

$$\text{COUNTBETWEEN}(a, b) \leftarrow \text{PARTIALSUM}(b) - \text{PARTIALSUM}(a).$$

If the return flag of TREESEARCH(a) is "found," the equation is the same as before, i.e.,

$$\text{COUNTBETWEEN}(a, b) \leftarrow \text{PARTIALSUM}(b) - \text{PARTIALSUM}(a) + x_a.num\_tags.$$