CSC265 Fall 20120 Homework Assignment 1

The list of people with whom I discussed this homework assignment: Jiawei Chen

A *rotated list* is a list that can be sorted by cyclically rotating the elements of the list. For example, 1 3 5 7 8 and 7 8 1 3 5 are rotated lists, but 1 7 3 5 8 is not.

The data structure *list of rotated lists* can be used to implement a dictionary of distinct keys. It consists of an array $A[1..s]$, and two integers n and g. The maximum size of the dictionary is s. The $n \leq s$ elements currently stored in the array are partitioned into g groups. Each group is maintained as a rotated list. The first group is stored in the first array element, $A[1]$. The second group is stored in the next two array elements, $A[2..3]$. The third group is stored in the next three array elements, $A[4..6]$. In general, for $1 \leq k < g$, the k^{th} group contains k consecutive elements of the array. However, the last group may contain fewer elements. Finally, the elements of each group are all less than the elements of the next group.

1. For $1 \leq k \leq g$, what is the location, $f(k)$, of the first element in group k? Briefly justify your answer.

   for the first element in group k, there are k-1 complete groups before it, each with the number of elements equal to its group number. This is $\sum_{i=1}^{k-1} i$ elements, plus 1 element for the first element in k-th group. $f(k) = 1 + 2 + 3 + ... + (k-1) + 1 = (k(k-1)/2) + 1 = (k^2 - k + 2)/2$. From this equation we can deduce a useful fact for the next questions. For any array with n elements, since $f(k) = n = (k^2 - k + 2)/2$, the number of groups with respect to n is in the order of sqrt(n).

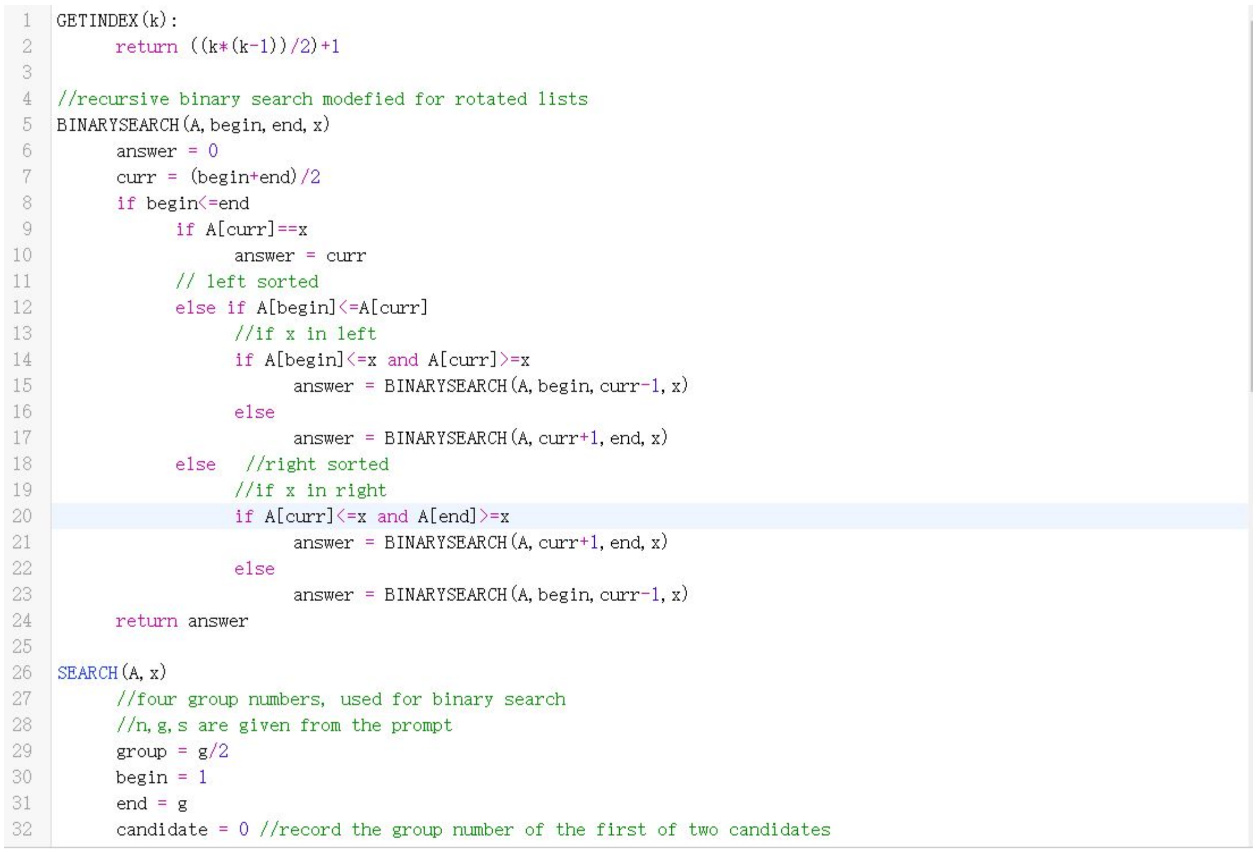
2. Give an algorithm (in pseudocode) for performing SEARCH(A, x) in this data structure. It should return the location i of x in $A[1..n]$ or 0, if x is not in $A[1..n]$. If it improves clarity, you can break up your algorithm into subprogams. Briefly explain how your algorithm works. Give the high level idea, NOT a line by line description of the pseudocode.

   the 70-line search pseudocode is in the following 2 pages, with detailed comments. (I included images, zooming large might help if it is unclear...please forgive me this time and tell me if you dont like this way...)

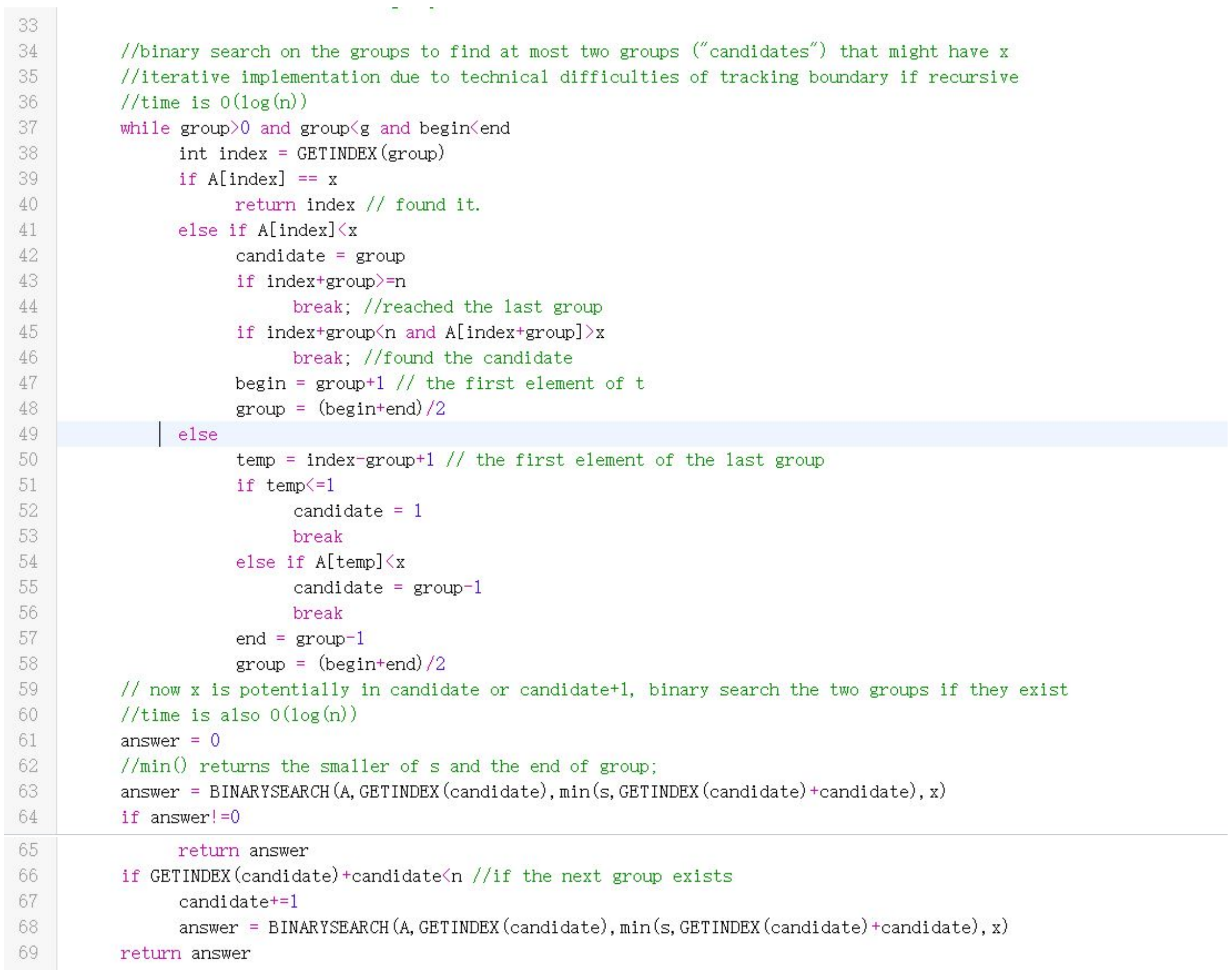
   Define min(a,b) to return the smaller of the two elements, where a,b are integers.

   Here is the high level idea: we first prove a lemma: assume there are two consecutive groups k and k+1 such that the first element of k (call it num1) is smaller than x and the first element of k+1 (num2) is larger than x. Assume x is in the array. Then, since every element in all groups before k are smaller than num1, and every element in all groups after k+1 are larger than num2, x must be in either group k or k+1. I name those groups candidates.

   The algorithm is at most 3 binary searches. We first binary search on the groups, and compare first elements with x, to find at most 2 candidates. This takes logn time. Then, after we found the candidates, we run a modified binary search on each of the candidates. Since rotated lists are always half-sorted in the middle(either left or right sublist is sorted), we can always compare x with the head and tail to find with half-list x is in, therefore eliminating half of the list each time (detail in comments). Each search takes logn time. Now x is found. Since the two steps are separate, the total runtime is still logn.



```
GETINDEX(k):
    return ((k*(k-1))/2)+1

//recursive binary search modefied for rotated lists
BINARYSEARCH(A,begin,end,x)
    answer = 0
    curr = (begin+end)/2
    if begin<=end
        if A[curr]==x
            answer = curr
        // left sorted
        else if A[begin]<=A[curr]
            //if x in left
            if A[begin]<=x and A[curr]>=x
                answer = BINARYSEARCH(A,begin,curr-1,x)
            else
                answer = BINARYSEARCH(A,curr+1,end,x)
        else   //right sorted
            //if x in right
            if A[curr]<=x and A[end]>=x
                answer = BINARYSEARCH(A,curr+1,end,x)
            else
                answer = BINARYSEARCH(A,begin,curr-1,x)
    return answer

SEARCH(A,x)
    //four group numbers, used for binary search
    //n,g,s are given from the prompt
    group = g/2
    begin = 1
    end = g
    candidate = 0 //record the group number of the first of two candidates
```



```
33
34      //binary search on the groups to find at most two groups ("candidates") that might have x
35      //iterative implementation due to technical difficulties of tracking boundary if recursive
36      //time is O(log(n))
37      while group>0 and group<g and begin<end
38          int index = GETINDEX(group)
39          if A[index] == x
40              return index // found it.
41          else if A[index]<x
42              candidate = group
43              if index+group>=n
44                  break; //reached the last group
45              if index+group<n and A[index+group]>x
46                  break; //found the candidate
47              begin = group+1 // the first element of t
48              group = (begin+end)/2
49          else
50              temp = index-group+1 // the first element of the last group
51              if temp<=1
52                  candidate = 1
53                  break
54              else if A[temp]<x
55                  candidate = group-1
56                  break
57              end = group-1
58              group = (begin+end)/2
59      // now x is potentially in candidate or candidate+1, binary search the two groups if they exist
60      //time is also O(log(n))
61      answer = 0
62      //min() returns the smaller of s and the end of group;
63      answer = BINARYSEARCH(A,GETINDEX(candidate),min(s,GETINDEX(candidate)+candidate),x)
64      if answer!=0
65          return answer
66      if GETINDEX(candidate)+candidate<n //if the next group exists
67          candidate+=1
68          answer = BINARYSEARCH(A,GETINDEX(candidate),min(s,GETINDEX(candidate)+candidate),x)
69      return answer
```

3. Prove that the worst case time complexity of your algorithm is $\Theta(\log n)$.

   A rough idea was given in the code comments and algorithm description.

   I will not prove O(logn) and $\Omega(\log n)$, but instead use the fact that binary search is $\Theta(\log n)$ from textbook page 39, and prove that the algorithm is composed of binary searches and O(1) operations.

   Consider binary search, we know that its runtime is $\Theta(\log n)$. Consider SEARCH. the while loop in lines 37-58 is essentially a binary search, except for the four conditional statements in lines 43,45,51 and 54. When any of these conditions are triggered, the loop exits after O(1) time. Therefore, in the worst case, none of those are triggered and the loop runs until the loop invariant is not satisfied. In this case, the loop is a regular binary search and therefore runs in $\Theta(\log n)$. Then, after some O(1) operations, SEARCH calls BINARYSEARCH at most twice, which we will show also runs in $\Theta(\log n)$ time.

   Consider BINARYSEARCH. This is a modified binary search.For each recursion, either it terminates directly or executes lines 15/17/21/23. Either begin = (begin+end)/2+1 or end = (begin+end)/2 -1, and this new begin/end is fed again to BINARYSEARCH at most once. In both cases, half of the input is discarded, so it takes at most logn+1 times to reduce the input, which is at most size n, to the base case. therefore, in the worst case, it takes logn+1 steps, which is $\Theta(\log n)$.

Since the algorithm is composed of at most 3 binary searches, each with $\Theta(\log n)$ runtime, and that there is no outer loops for each of the searches, the worst runtime is simply the sum of the three binary searches which is still $\Theta(\log n)$.

4. Prove that your SEARCH algorithm is correct. Note that you may need to state and prove some additional lemmas.

   Given input (A,x) where x is in A,consider the while loop in line 37. Let mid = (begin+end)/2. The invariant is that while the loop executes, group is between 0 and g, and that begin is always smaller or equal to end.

   initialization: This condition initially holds, since begin = 1 ¡ end = n. group is g/2, which is between 0 and g.

   output correct: after each iteration, (mid+1+end)/2 in line 48 is no larger than end, and (mid-1+begin)/2 is no smaller than begin, which can be proven by substituting mid. If x is larger than mid, then by lines 47-48, mid shifts right; if x smaller, then mid shifts left by lines 57-58.

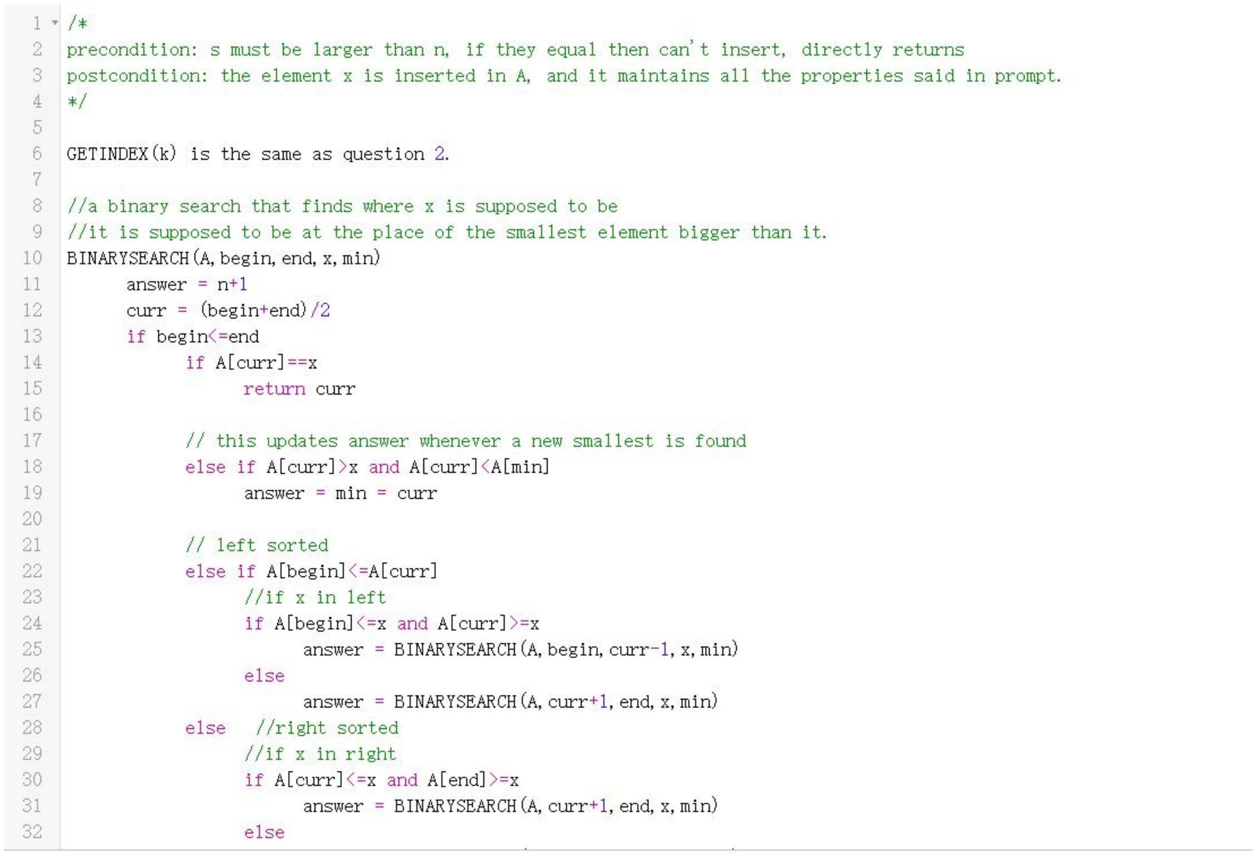
   This keeps happening until the boundary is reached or line 45/54 is triggered. If x is in last group, line 43-44 returns last group only as expected. If x is in first group, line 51-53 returns first and second group as expected. If line 45 is triggered, then the current group's first is smaller than x, and x has next group which has a greater first element, so the current index and the next are candidates; similarly, if line 54 is triggered, then current group first is larger than x and last group first is smaller than x, so they are candidates.

   Therefore, the first of at most two candidates is always returned. If x is in first candidate, then line 63 uses binary search within the group to find it, which in returned in line 65 as correct output. If the second candidate exists, then line 68 searches within the group to find x, which if found is correctly returned in line 70. If x is not found in both candidates, then answer remains 0, which is correctly returned in line 70.

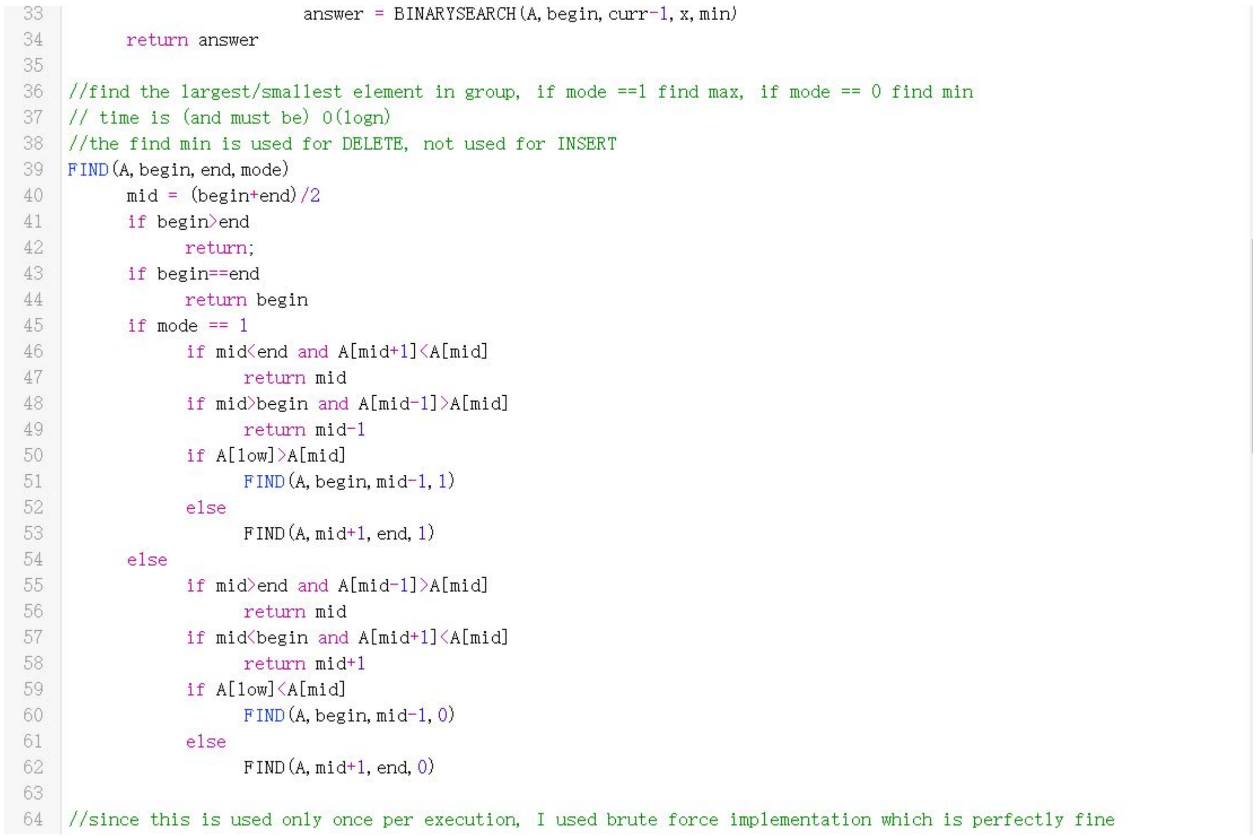
   termination: GETINDEX obviously terminates. For BINARYSEARCH, it terminates when begin>end or A[curr]==x. In other cases, since begin¡=end, we know curr$\geq$begin and curr$\leq$end. So curr+1>begin and curr-1<end. Either line 15,17,21 or 23 is recursively called. In all four cases, either begin increases by at least 1 or end decreases by at least 1. Since end-begin is at most n, after at most n iterations, BINARYSEARCH must terminate.

   For SEARCH itself, lines 61-70 must terminate because both functions it calls terminates. for the while loop, it terminates when group is not in [0,1,...,g], or when begin$\geq$end, or when x is found. Assume x is not found and while does not terminate and the break statements are not triggered. Similar to last paragraph, we know group$\geq$ begin and group$\leq$end. group+1>begin and group-1<end. For each iteration, either begin increases by at least 1 or end decreases by at least one, with the other variable unchanged. So the while loop runs at most g times. Therefore, SEARCH terminates.

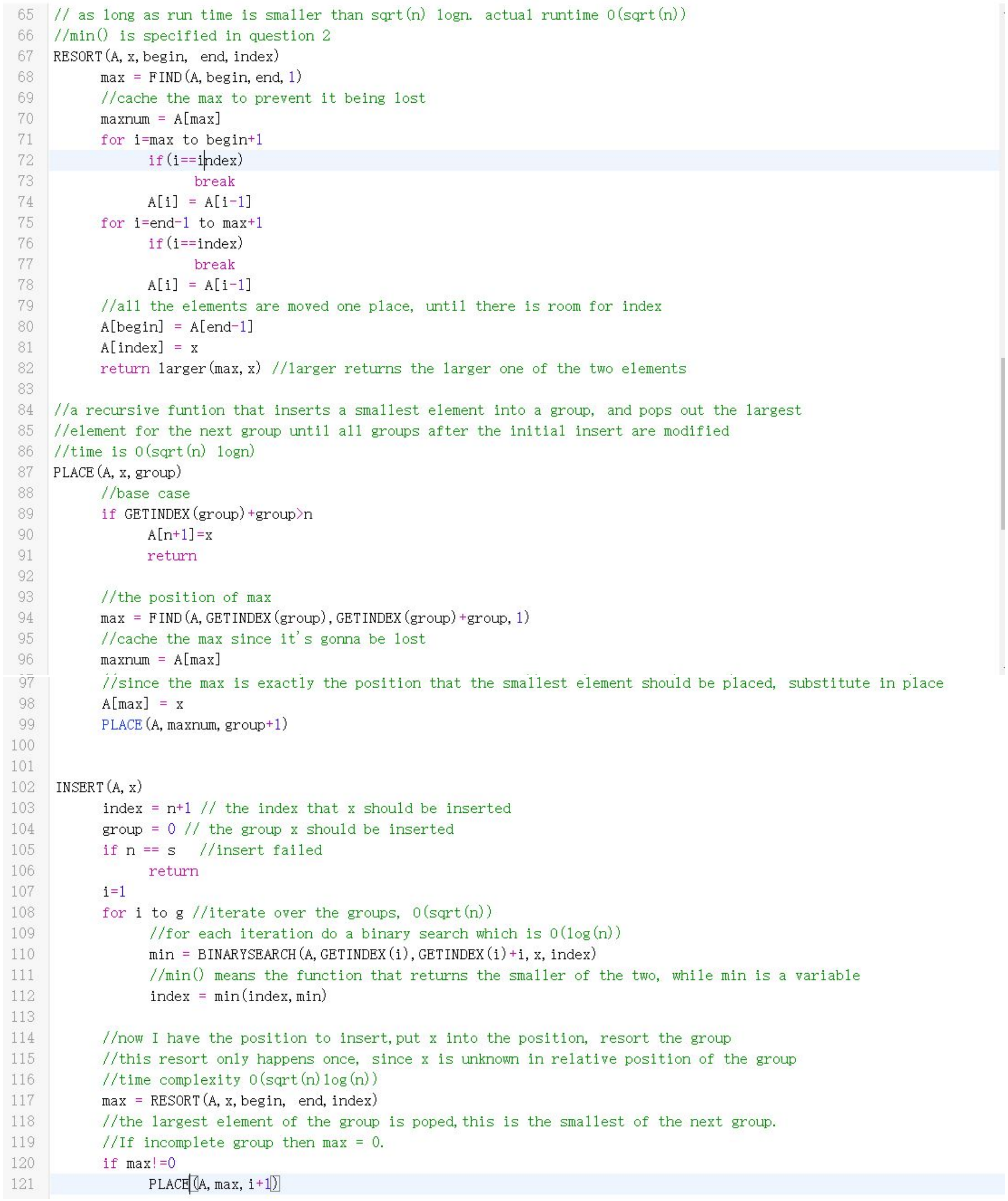
5. Give an algorithm (in pseudocode) for performing INSERT in this data structure in $O(\sqrt{n}\log n)$ time. Give precise specifications for your algorithm. Briefly explain how your algorithm works, why it is correct, and why it runs in $O(\sqrt{n}\log n)$ time.



```
/*
precondition: s must be larger than n, if they equal then can't insert, directly returns
postcondition: the element x is inserted in A, and it maintains all the properties said in prompt.
*/

GETINDEX(k) is the same as question 2.

//a binary search that finds where x is supposed to be
//it is supposed to be at the place of the smallest element bigger than it.
BINARYSEARCH(A, begin, end, x, min)
      answer = n+1
      curr = (begin+end)/2
      if begin<=end
            if A[curr]==x
                  return curr

            // this updates answer whenever a new smallest is found
            else if A[curr]>x and A[curr]<A[min]
                  answer = min = curr

            // left sorted
            else if A[begin]<=A[curr]
                  //if x in left
                  if A[begin]<=x and A[curr]>=x
                        answer = BINARYSEARCH(A, begin, curr-1, x, min)
                  else
                        answer = BINARYSEARCH(A, curr+1, end, x, min)
            else   //right sorted
                  //if x in right
                  if A[curr]<=x and A[end]>=x
                        answer = BINARYSEARCH(A, curr+1, end, x, min)
                  else
```



```
                        answer = BINARYSEARCH(A, begin, curr-1, x, min)
        return answer

//find the largest/smallest element in group, if mode ==1 find max, if mode == 0 find min
// time is (and must be) O(logn)
//the find min is used for DELETE, not used for INSERT
FIND(A, begin, end, mode)
        mid = (begin+end)/2
        if begin>end
            return;
        if begin==end
            return begin
        if mode == 1
            if mid<end and A[mid+1]<A[mid]
                return mid
            if mid>begin and A[mid-1]>A[mid]
                return mid-1
            if A[low]>A[mid]
                FIND(A, begin, mid-1, 1)
            else
                FIND(A, mid+1, end, 1)
        else
            if mid>end and A[mid-1]>A[mid]
                return mid
            if mid<begin and A[mid+1]<A[mid]
                return mid+1
            if A[low]<A[mid]
                FIND(A, begin, mid-1, 0)
            else
                FIND(A, mid+1, end, 0)

//since this is used only once per execution, I used brute force implementation which is perfectly fine
```



```
// as long as run time is smaller than sqrt(n) logn. actual runtime O(sqrt(n))
//min() is specified in question 2
RESORT(A, x, begin,  end, index)
    max = FIND(A, begin, end, 1)
    //cache the max to prevent it being lost
    maxnum = A[max]
    for i=max to begin+1
        if(i==index)
            break
        A[i] = A[i-1]
    for i=end-1 to max+1
        if(i==index)
            break
        A[i] = A[i-1]
    //all the elements are moved one place, until there is room for index
    A[begin] = A[end-1]
    A[index] = x
    return larger(max, x) //larger returns the larger one of the two elements

//a recursive funtion that inserts a smallest element into a group, and pops out the largest
//element for the next group until all groups after the initial insert are modified
//time is O(sqrt(n) logn)
PLACE(A, x, group)
    //base case
    if GETINDEX(group)+group>n
        A[n+1]=x
        return

    //the position of max
    max = FIND(A, GETINDEX(group), GETINDEX(group)+group, 1)
    //cache the max since it's gonna be lost
    maxnum = A[max]
    //since the max is exactly the position that the smallest element should be placed, substitute in place
    A[max] = x
    PLACE(A, maxnum, group+1)


INSERT(A, x)
    index = n+1 // the index that x should be inserted
    group = 0 // the group x should be inserted
    if n == s   //insert failed
        return
    i=1
    for i to g //iterate over the groups, O(sqrt(n))
        //for each iteration do a binary search which is O(log(n))
        min = BINARYSEARCH(A, GETINDEX(i), GETINDEX(i)+i, x, index)
        //min() means the function that returns the smaller of the two, while min is a variable
        index = min(index, min)

    //now I have the position to insert, put x into the position, resort the group
    //this resort only happens once, since x is unknown in relative position of the group
    //time complexity O(sqrt(n)log(n))
    max = RESORT(A, x, begin,  end, index)
    //the largest element of the group is poped, this is the smallest of the next group.
    //If incomplete group then max = 0.
    if max!=0
        PLACE(A, max, i+1)
```

high level idea: first binary search to find smallest element larger than x or just n+1. This is the place to insert. name this group j. Inserting x is special, so manually move and resort elements to fit x, and pop out the max(which can be x). from the next group until the last group(which might not exist for now), since the previous max is the smallest of this group, find the new max, replace it with the previous max, and now the new max is previous max. Recursively do this for the next group until the end is reached.

runtime: GETINDEX is O(1). BINARYSEARCH is O(logn), because if it does not terminate early, each time the input is reduced by at least a half. FIND is also a binary search if it does not terminate early, so it is O(logn).

RESORT first calls FIND, which is O(logn). then, it uses for loop on at most end-begin elements,each with O(1) time, which in question 1 proved that there are O(sqrt(n)) elements. then it returns the larger of max and x, which is also O(1). Overall, this is O(sqrt(n))+O(logn) = O(sqrt(n)) time.

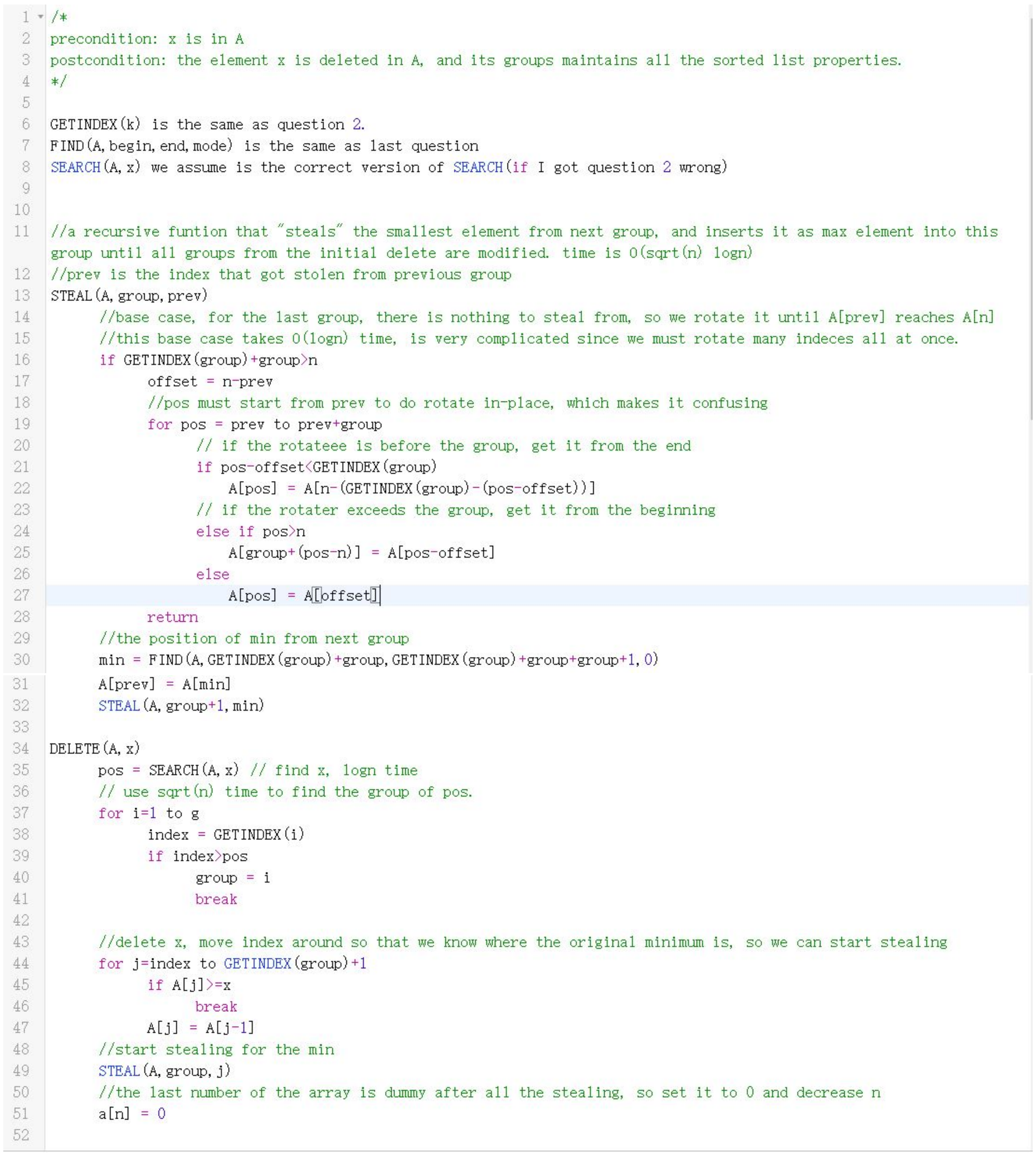
PLACE recurs over the groups, which is at most O(sqrt(n)) recursions. for each recursion, some O(1) operations and a FIND is called, making it O(logn) runtime. Overall, the runtime is $O(\sqrt{n}\log n)$.

Finally for INSERT. it first iterates over the groups, for each iteration it calls BINARYSEARCH, this is $O(\sqrt{n}\log n)$ runtime. Then, it calls RESORT once, which is O(sqrt(n)) time. Then, it calls PLACE at most once, which is $O(\sqrt{n}\log n)$ runtime. Overall, the runtime is $O(\sqrt{n}\log n)$ + O(sqrt(n)) + $O(\sqrt{n}\log n)$, which is still $O(\sqrt{n}\log n)$.

When replacing the maximum of a rotated list with an number smaller than all its elements, the list is still a rotated list. Output correct:assume we want to insert an element x. in the loop from line 108-112, the index of the minimum element greater than x is recorded. If such an element does not exist, then index is by default n+1, which means appending to the end. This is all correct output. Then RESORT is called and x is inserted in the correct location, while that group maintains the sorted list property. If this causes the group to have extra elements, the largest is poped out, and then recursively inserted in the following groups. In the end, each group maintains its rotated list property, and x is inserted in the correct position.

termination:BINARYSEARCH and FIND terminates because they are both binary search, and the input shrinks by at least half for each recursion. RESORT terminates because FIND terminates, and it additionally loops on finite elements which also terminates. PLACE terminates because FIND terminates and for each iteration, the group number is increased by 1, making there at most g iterations. Finally, INSERT terminates because it first iterates on a finite loop which calls BINARYSEARCH each iteration, then it calls RESORT and PLACE. Since all those functions terminate, INSERT terminates as well.

6. Give an algorithm (in pseudocode) for performing DELETE in this data structure in $O(\sqrt{n}\log n)$ time. Give precise specifications for your algorithm. Briefly explain how your algorithm works, why it is correct, and why it runs in $O(\sqrt{n}\log n)$ time.



```
/*
precondition: x is in A
postcondition: the element x is deleted in A, and its groups maintains all the sorted list properties.
*/

GETINDEX(k) is the same as question 2.
FIND(A,begin,end,mode) is the same as last question
SEARCH(A,x) we assume is the correct version of SEARCH(if I got question 2 wrong)


//a recursive funtion that "steals" the smallest element from next group, and inserts it as max element into this group until all groups from the initial delete are modified. time is O(sqrt(n) logn)
//prev is the index that got stolen from previous group
STEAL(A,group,prev)
    //base case, for the last group, there is nothing to steal from, so we rotate it until A[prev] reaches A[n]
    //this base case takes O(logn) time, is very complicated since we must rotate many indeces all at once.
    if GETINDEX(group)+group>n
        offset = n-prev
        //pos must start from prev to do rotate in-place, which makes it confusing
        for pos = prev to prev+group
            // if the rotateee is before the group, get it from the end
            if pos-offset<GETINDEX(group)
                A[pos] = A[n-(GETINDEX(group)-(pos-offset))]
            // if the rotater exceeds the group, get it from the beginning
            else if pos>n
                A[group+(pos-n)] = A[pos-offset]
            else
                A[pos] = A[offset]
        return
    //the position of min from next group
    min = FIND(A,GETINDEX(group)+group,GETINDEX(group)+group+group+1,0)
    A[prev] = A[min]
    STEAL(A,group+1,min)

DELETE(A,x)
    pos = SEARCH(A,x) // find x, logn time
    // use sqrt(n) time to find the group of pos.
    for i=1 to g
        index = GETINDEX(i)
        if index>pos
            group = i
            break

    //delete x, move index around so that we know where the original minimum is, so we can start stealing
    for j=index to GETINDEX(group)+1
        if A[j]>=x
            break
        A[j] = A[j-1]
    //start stealing for the min
    STEAL(A,group,j)
    //the last number of the array is dummy after all the stealing, so set it to 0 and decrease n
    a[n] = 0

```

high level idea: search for the position of x. shift all group elements to the left of x that is smaller to it 1 index to the right until we found the minimum of the group. Then, "steal" the minimum element from the next group and replace the original minimum, because the stolen element is the maximum in this group. Do this until there is no more to steal, then shift the last group until the stolen element is at last, then set it to 0.

Runtime: STEAL recurs at most sqrt(n) times, because each iteration it steals from the next group, and the number of groups is O(sqrt(n)). For each iteration, it calls FIND once, which we proved is O(logn) from question 5. In the base case, STEAL shifts every element in the last group exactly once, so it is also sqrt(n) runtime. Therefore, the runtime of STEAL is $O(\sqrt{n}\log n)$.

DELETE first spends O(logn) time by calling SEARCH. Then, it spends sqrt(n) time by iterating over the groups to find the group x belongs to. In order to find the original minimum, line 44 iterates at most O(sqrt(n)) times because that group only has at most O(sqrt(n)) elements. At last, it calls STEAL, which is $O(\sqrt{n}\log n)$ time. Overall, DELETE takes O(logn)+O(sqrt(n))+O(sqrt(n))+$O(\sqrt{n}\log n)$ time, which is still $O(\sqrt{n}\log n)$.

Correctness: When replacing the minimum of a rotated list with an number larger than all its elements, the list is still a rotated list.

output correct: assume we want to delete x from A. We first find the position of x, and then shifts group elements to the left and smaller than x by 1 to the right, so that the minimum is found. we then get the minimum from next group to replace it, so that the group is still a rotated list. For each stealing, the group after replacement is always a rotated list, so the data structure maintains its integrity. This repeats until in the end, we rotate the last group such that the stolen element is at last and reset, so the last group maintains integrity as well. Therefore, all groups in A maintains its rotated list property, and x has successfully been deleted.

termination:We know GETINDEX, FIND, SEARCH terminates. For STEAL, for each recursion, the group number increases by 1 until it reaches the base case(last group). Therefore, STEAL terminates. For DELETE, the for loop in line 37 terminates because it iterates over finite elements and executes O(1) command. the for loop on line 44 terminates for the same reason. STEAL terminates. Therefore, DELETE also terminates.

7. What are the advantages and disadvantages of using this data structure to implement a DICTIONARY as compared to using a sorted array and an unsorted array?

   This structure is very slow when the data size is small. Assume that there are sufficiently large data. This structure has O(logn) search and $O(\sqrt{n}\log n)$ insert and delete time, while sorted array has O(logn) search and O(n) insert and delete time, and unsorted array has O(n) search time and O(1) insert and delete time. Compared to sorted array, this structure has asymptotically better insert/delete time and same search time. Compared to unsorted array, this structure has asymptotically better search time and worse insert/delete time. Therefore, when both search and insert/delete operations are frequently required for a large data set, this structure has the best performance among the three.