CSCI-GA.1170-001/002 Fundamental Algorithms October 14, 2015

Solutions to Problem 1 of Homework 4 (10 points)

Name: GOWTHAM GOLI (N17656180) *Due: Tuesday, October 6*

Using a min-heap in a clever way, give $O(n \log k)$-time algorithm to merge k sorted arrays $A_1 \ldots A_k$ of size n/k each into one sorted array B. Write the pseudocode of your algorithm using procedures BUILD-HEAP, EXTRACT-MIN and INSERT.

Solution:

Algorithm

- Build a Heap using the first elements of the given arrays $A_1, \ldots, A_k$
- Extract Minimum from the heap and insert into B
- Pick the next element from the array, the popped element of the heap came from and insert into this heap
- Keep repeating until all the elements of the given arrays are inserted into the heap

Psuedocode

Algorithm: MERGEHEAPS($A_1, \ldots, A_k, B$)
Heap $\leftarrow$ BUILD-HEAP($A_1[0], \ldots, A_k[0]$)
$B \leftarrow$ NEWARRAY(n)
$c \leftarrow 0$
while *Heap is not empty* **do**
    $A_i[p] \leftarrow =$ EXTRACT-MIN(*Heap*)
    $B[c] \leftarrow A_i[p]$
    if $p < n/k$ **then**
        INSERT($A[p+1], Heap$)
    end
    c++
end
Return B

Algorithm 1: Merge sorted arrays $A_1, \ldots, A_k$ using a min-heap

Time Complexity

In the first step, time taken to build heap of size k is $O(k)$. Time taken to extract minimum from the heap and then insert into the heap is $O(\log k)$ and this is done n times, therefore $O(n \log k)$ Therefore, total time takes is $O(k) + O(n \log k) = O(n \log k)$

$\square$

CSCI-GA.1170-001/002 Fundamental Algorithms	October 14, 2015

Solutions to Problem 2 of Homework 4 (26 (+14) Points)

Name: GOWTHAM GOLI (N17656180) *Due: Tuesday, October 6*

Your are given an array $A[1]\dots A[n]$ of n "objects". You have a magic unit-time procedure $Equal(A[i], A[j])$, which will tell if objects $A[i]$ and $A[j]$ are the same. Unfortunately, there is no other way to get any meaningful information about the objects: e.g., cannot ask if $A[i]$ is "greater" than $A[j]$ of if it is more "sexy", etc., just the equality test. We say that A is a *repetitor* if it contains strictly more than $n/2$ elements which are all pairwise the same. In this case any of A's (at least $n/2$) repetitive elements is called *dull*. For example, if the "object" is a string, the array $(boring, funny, cute, boring, boring)$ is a repetitor where $boring$ is dull while $funny$ is not. On the other hand, the array $(hello, hi, bonjorno, hola, whasup)$ is not a repetitor. You goal is to determine if A is a repetitor, and, if so, output its dull "object" (which is clearly unique).

(a) (8 points) Design a simple divide-and-conquer algorithm for this problem running in time $O(n \log n)$. Make sure you argue the correctness and the running time.
(**Hint**: Prove that if A is a repetitor, at least one of its "halves" is as well.)

Solution:

Theorem 1. *If A is a repetitor, at least one of its "halves" is as well*

Proof. Let p be the dull of A. If the left half of A is not a repetitor then there are less than $n/4$ occurrences of p in $A[1\dots mid]$ which implies that the right half $A[mid+1\dots n]$ has to have more than $n/4$ occurrences of p for p to be a dull of A. Therefore, the right half is a repetitor. Similarly, we can argue for the left half to be a repetitor.

If neither of the halves of A is a repetitor, it means that there are less than $n/4$ occurrences of p in $A[1\dots mid]$ and $A[mid+1\dots n]$. Therefore, p cannot be the dull in this case

If both the halves of A are repetitors then $A[1\dots mid]$ has to have more than $n/4$ occurrences of p and $A[mid+1\dots n]$ has to have more than $n/4$ occurrences of p for p to be a dull of A.

Therefore we can conclude that If A is a repetitor, at least one of its "halves" is as well $\square$

Using the above theorem, we can implement a $O(n \log n)$ algorithm as follows

Psuedocode

```
Algorithm: FindDull(A, low, high)
if low is high then
    return A[low]
end
else
    mid ← (low + high)/2
    leftDull ← FindDull(A, low, mid)
    rightDull ← FindDull(A, mid + 1, high)
    if leftDull is rightDull then
        Return leftDull
    end
    else
        lDullCount = CountOccurrences(leftDull, A[low ... high])
        rDullCOunt = CountOccurrences(rightDull, A[low ... high])
        if lDullCount > (high - low)/2 then
            Return leftDull
        else if rDullCount > (high - low)/2 then
            Return rightDull
        else
            Return NoDull
    end
end
```

Algorithm 2: Alorithm to find the dull OF A in $O(n \log n)$ time

□

(b) (4 Points) Remember, if A was an integer array, the procedure PARTITION(A, p, r) (see Section 7.1) makes $x = A[r]$ the pivot element and returns the index q, where the new value of $A[q]$ contains the pivot x, the new values $A[p \dots q-1]$ contain elements less or equal to x, and the new values $A[q+1 \dots r]$ contain values greater than x. Write the pseudocode of the modified procedure NEW-PARTITION(A, p, r), which only uses the *Equal* operator and returns q such that $A[q+1 \dots r]$ contain all the elements equal to x (while $A[p \dots q]$ contain all other elements).

Solution:

Psuedocode

Algorithm: NEW-PARTITION(A, p, r)
$x \leftarrow A[r]$
$i \leftarrow p - 1$
for $j = p$ *to r-1* **do**
  if *A[j]* NOT EQUAL x **then**
    $i \leftarrow i + 1$
    SWAP$(A[i], A[j])$
  end
end
SWAP$(A[i+1], A[r])$
Return $i + 1$

Algorithm 3: Modified procedure NEW-PARTITION(A, p, r)

□

(c) (2 points) Consider the following, more general, algorithm REPEAT(A, n, t), which tells if some element of $A[1] \ldots A[n]$ is repeated at least t times. (Clearly, REPETITOR can just call REPEAT with $t = n/2 + 1$.)

REPEAT$(A,\ n,\ t)$
  If $n < t$ **Return** *no*
  Pick $i \in \{1 \ldots n\}$ *at random.*
  Swap$(A[i], A[n])$
  $q \leftarrow$NEW-PARTITION$(A,\ 1,\ n)$
  If $n - q \geq t$ **Then Return** $(yes, A[n])$
  Return REPEAT$(A,\ q,\ t)$

Argue that the algorithm above is correct.

Solution:

It is quite obvious that $t \leq n$, Therefore if $t > n$ the algorithm returns *no*

Let the randomly picked element be *key*. NEW-PARTITION$(A, 1, n)$ returns A such that $A[1 \ldots q-1] \neq key$ and $A[q \ldots n] = key$. Therefore the array has $n - q$ occurrences of *key*

If $n - q \geq t$ then *key* is repeated atleast t times. Else we keep repeating the procedure with a new key each time until we find an element that is repeated atleast t times □

(d) (3 points) Argue that the algorithm above always terminates in time $O(n^2)$ (irrespective of the random choices of i).

Solution: In the worst case scenario, If the array has all distinct elements and no element can occurs more than t times ($\because t \geq 2$), q will be $n-1$ in the first recursive call, $n-2$ in the second recursive call, $n-3$ in the third recursive and so on until 1 in the first recursive call

Therefore in the worst case, the running time $T(n)$ is

$$\begin{aligned}
T(n) &= T(n-1) + n \\
&= T(n-2) + n - 1 + n \\
&= T(n-3) + n - 2 + n - 1 + n \\
&\vdots \\
&= 1 + 2 + \ldots + n - 1 + n \\
&= \frac{n(n+1)}{2} \\
&= O(n^2)
\end{aligned}$$

□

(e) (3 points) Give an example of an (integer) array A and a value $t \geq 2$ where the algorithm indeed takes time $\Omega(n^2)$.

Solution: Let $A = [1, 2, 3, 4, 5, 6, 7, 8, 9]$ and $t = 3$. There is no element in A that occurs more than 3 times. The algorithm will run as follows

Irrespective of the element picked as the pivot, as all the elements are distinct, q will be $n - 1$ (i.e 8) in the first recursive call, $n - 2$ (i.e 7) in the second recursive call and so on until 1 in the first recursive call.

$\implies T(n) = 1 + 2 + \ldots + n = n(n+1)/2 = \Omega(n^2)$ □

(f) (4 Points) Let $T(n)$ be the worst case (over all arrays $A[1 \ldots n]$ and $t > n/2$ such that A contains t identical elements) of the *expected* running time of $\textsc{Repeat}(A, n, t)$ (over the random choice of i). For concreteness, assume $\textsc{New-Partition}$ takes time exactly n. Prove that

$$T(n) \leq \frac{1}{2} \cdot T(n-1) + n$$

(**Hint**: Prove that in this case no recursive sub-call will be made with probability $t/n > 1/2$.)

Solution: Given that A contains t identical elements ($t > n/2$), Therefore A is repititor and let p be it's dull.

If p is randomly picked up as the pivot, then the algorithm will terminate after the first recursive call itself. Therefore the probability that there will be no recursive sub call is the (total number of occurrences of p)/(number of elements in A) $> (n/2n) = 1/2$

In the worst case, For a recursive call to happen, the pivot chosen has to be a unique non-dull (or the element with least number of repetitions if the elements are not distinct) element of A. Probability that the non-dull element will be chosen as a pivot is $< 1/2$

Therefore in the worst case of the expected running time,
$T(n) =$ (probability that the next recursive call will happen)(Time taken by the next recursive call) $+ n$
$\implies T(n) \leq \frac{1}{2}.T(n-1) + n$ □

(g) (2 points) Show by induction that $T(n) \leq 2n$.

Solution:

Base case - $T(1) = 1 \leq 2.1 = 2$. Base case is true

Induction Hypothesis - $T(k) \leq 2k, \ \forall k = 1, \ldots, n-1$

Induction Step

$$\begin{aligned} T(n) &\leq \frac{1}{2}.T(n-1) + n \\ &\leq \frac{1}{2}(2(n-1)) + n \\ &= n - 1 + n \\ &\leq 2n \end{aligned}$$

By Induction, we can conclude that $T(n) \leq 2n$ □

(h*) (**Extra Credit**; 6 points) Consider the following test for repetitor. For 100 times, run $\text{Repeat}(A, n, n/2+1)$ for at most $4n$ steps. If one of these 100 runs ever finishes within $4n$ steps, use that answer. If none of the 100 runs terminates within $4n$ steps, return *no*. Argue that the running time of this procedure is $O(n)$. Then argue that the probability it returns the incorrect *no* answer (when it should have returned *yes*) is at most 2^{-100}.
(**Hint**: Show that when the answer is *yes*, the probability of not finding this answer in $4n$ steps is at most 1/2. Google for "Markov's inequality" if you want to be formal.)

Solution: The test runs for at most $4n$ steps and 100 times, therefore total number of steps is $400n$. Therefore running time is $O(n)$ ($\because 400 << n$) □

(i**) (**Extra Credit**; 8 points) Try to design $O(n)$ deterministic test for a repetitor.

Solution:

Key Idea

The dull of the array remains preserved whenever we cancel a pair of distinct elements from the array

Psuedocode

```
Algorithm: Find-Dull(A)
count ← 0
for i ← 0 to n-1 do
    if count is 0 then
        x ← A[i]
        count ← 1
    else if x ≠ A[i] then
        count ← count − 1
    Else count ← count + 1
    num ← Count-Occurrences(x, A)
    if num > n/2 then
        Return x
    else
        Return NoDull
end
```

Algorithm 4: Algorithm to find dull of A in $O(n)$ time

To prove the correctness of the above algorithm it is sufficient if we prove that if A is a repititor and say α is the dull then at the end of the for loop $x = \alpha$

In the above algorithm, during i^{th} iteration, we compare $A[i-1]$ with x and cancel both if they are different, and increment count otherwise.

So if *count* $= 0$, then all elements upto $A[i-1]$ would have been eliminated through distinct-elements pair formations. If *count* > 0, then $\{x, \ldots$ count times $\ldots, x, A[i], \ldots A[n-1]\}$ elements would have still survived at the end of the i^{th} iteration.
Let S_i be $\{x, \ldots$ count times $\ldots, x, A[i], \ldots A[n-1]\}$ then α is a dull of S_i.

At the end of for loop $S_n = \{x, \ldots$ count times $\ldots, x, A[n], \ldots A[n-1]\}$. It means that α is a dull of $\{x, \ldots$ count times $\ldots, x\}$ and *count* > 0. Hence $\alpha = x$ $\square$

CSCI-GA.1170-001/002 Fundamental Algorithms	October 14, 2015

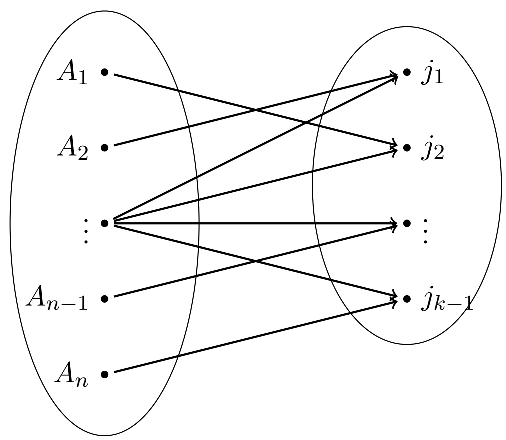
Solutions to Problem 3 of Homework 4 (12 points)

Name: GOWTHAM GOLI (N17656180) *Due: Tuesday, October 6*

Assume we are given an array $A[1 \dots n]$ of n *distinct* integers and that $n = 2k$ is *even*.

(a) (4 points) Let $pivot(A)$ denote the rank of the pivot element at the end of the partition procedure, and assume that we choose a random element $A[i]$ as a pivot, so that $pivot(A) = i$ with probability $1/n$, for all i. Let $smallest(A)$ be the length of the smaller sub-array in the two recursive subcalls of the QUICKSORT. Notice, $smallest(A) = \min(pivot(A) - 1, n - pivot(A))$ and belongs to $\{0 \dots k-1\}$, since $n = 2k$ is even. Given $0 \le j \le k-1$, what is the probability that $smallest(A) = j$?

Solution: Given that the probability to choose a random element $A[i]$ as a pivot is $1/n$. It is clear that there exists a pair $(A_i, A_j) \le n$ such that $A[i]$ and $A[j]$ will result in the same *smallest(A)* when chosen as a pivot. Therefore, there will be total $n/2$ pairs of (A_i, A_j) that will result in the same *smallest(A)* when chosen as a pivot which is illustrated in the figure below (j_k denotes the case when $j = k$)



As we know that the probability of selecting any A_i as pivot is $1/n$. The probability that $smallest(A)$ will be $2/n$ (as two elements from the Pivot set point to one element in the Smallest set as seen above).

$$\Pr(smallest(A) = j) = 2/n$$

$\square$

(b) (3 points) Compute the *expected value* of $smallest(A)$; i.e., $\sum_{j=0}^{k-1} \Pr(smallest(A) = j) \cdot j$. (**Hint**: If you solve part (a) correctly, no big computation is needed here.)

Solution:

$$\begin{aligned}\sum_{j=0}^{k-1} \Pr(smallest(A) = j) \cdot j &= \sum_{j=0}^{k-1} 2/n.j \\ &= 2/n \sum_{j=0}^{k-1} j \\ &= \frac{2}{n}.\frac{k.(k-1)}{2} \\ &= \frac{n-2}{4}\end{aligned}$$

□

(c) (5 points) Write a recurrence equation for the running time $T(n)$ of QUICKSORT, assuming that at every level of the recursion the corresponding sub-arrays of A are partitioned *exactly* in the ratio you computed in part (b). Solve the resulting recurrence equation. Is it still as good as the average case of randomized QUICKSORT?

Solution:

For relatively large n we can assume that $n - 2/4 \approxeq n/4$. Therefore at every level of the recursion the array is partitioned into sub arrays of length $n/4$ and $3n/4$.

$$T(n) = T(n/4) + T(3n/4) + O(n)$$

This recurrence equation can be solved using Recursive tree method as shown below in the figure

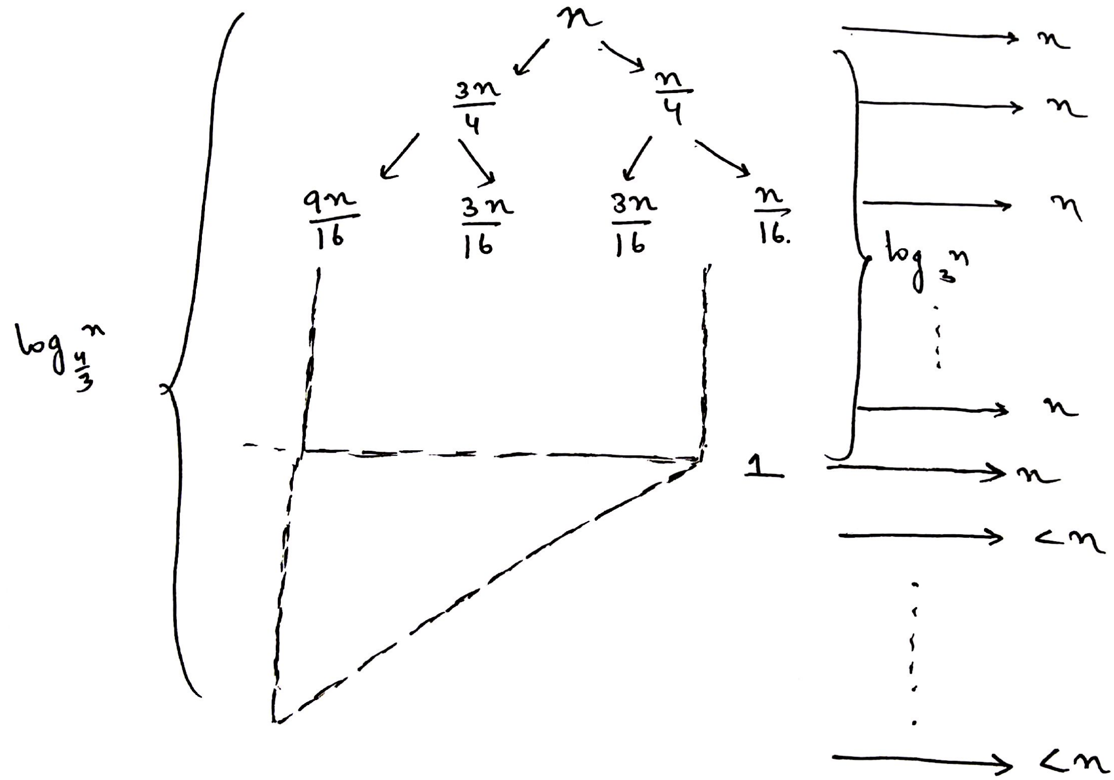


Figure 1: Recursive Tree of $T(n)$

$$
\begin{aligned}
n\log_4 n \le T(n) \le n\log_{4/3} n \\
n\frac{\log_2 n}{\log_2 4} \le n \le \frac{\log_2 n}{\log_2 4/3} \\
\implies T(n) = \Theta(n\log n)
\end{aligned}
$$

We can see that it as good as the average case of randomized quick sort $\square$

CSCI-GA.1170-001/002 Fundamental Algorithms October 14, 2015

Solutions to Problem 4 of Homework 4 (12 points)

Name: GOWTHAM GOLI (N17656180) *Due: Tuesday, October 6*

We wish to implement a data structure D that maintains the k smallest elements of an array A. The data structure should allow the following procedures:

- $D \leftarrow$ Initialize(A, n, k) that initializes D for a given array A of n elements.
- Traverse(D), that returns the k smallest elements of A *in sorted order*.
- Insert(D, x), that updates D when an element x is inserted in the array A.

We can implement D using one of the following data structures: (i) an unsorted array of size k; (ii) a sorted array of size k; (iii) a max-heap of size k.

(a) (4 points) For each of the choices (i)-(iii), show that the Initialize procedure can be performed in time $O(n + k \log n)$.

Solution:

Initialize

Approach

The idea is to find the k^{th} smallest element in the array A and then traverse the array to find all the elements less than the k^{th} smallest element.

Following the idea of Quicksort, the following algorithm is called Quickselect which returns the k^{th} smallest element in $O(n)$ time. We choose a random element as a pivot and now we know in which partition the k^{th} smallest element lies in. So we recursively find the desired element in that partition

Algorithm: Quickselect$(A, low, high, k)$
if *low is high* **then**
  Return A[low]
end
$q \leftarrow$ Partition$(A, low, high)$
if q *is* k **then**
  Return $A[q]$
else if $q > k$ **then**
  Return Quickselect$(A, low, q-1, k)$
else
  Return Quickselect$(A, q+1, high, k)$

Algorithm 5: Algorithm that returns k^{th} smallest element of A in $O(n)$ time

Time Complexity of Quickselect

$$\begin{aligned} T(n) &= T(q) + O(n) \;\; (or) \;\; T(n-q) + O(n) \;\; where \;\; q \leftarrow 1 \;\; to \;\; n-1 \\ &= T(k) + O(n) \;\; where \;\; k \leftarrow 1 \;\; to \;\; n-1 \end{aligned}$$

$$\therefore T(n) = \begin{cases} T(1) + O(n) \\ T(2) + O(n) \\ \vdots \\ T(n-1) + O(n) \end{cases}$$

Expected Running Time, $T(n)$ will be

$$\begin{aligned} T(n) &= \frac{1}{n}(T(0) + T(1) + \ldots + T(n-1) + n^2) \\ nT(n) &= T(0) + T(1) + \ldots + T(n-1) + n^2 \\ (n-1)T(n-1) &= T(0) + T(1) + \ldots + T(n-2) + (n-1)^2 \\ nT(n) - (n-1)T(n-1) &= T(n-1) + 2n - 1 \\ nT(n) &= nT(n-1) + 2n - 1 \\ T(n) &= T(n-1) + 2 - 1/n \\ &= 2n - (1/n + 1/(n-1) + \ldots + 1) \\ &\leq 2n \\ &= O(n) \end{aligned}$$

Therefore the average running time of Quickselect is only $O(n)$. However in the worst case, it could go to $O(n^2)$

i Using Unsorted array

```
Algorithm: Initialize(A)
D ← newArray(k)
c ← 0
key ← Quickselect(A, 0, n − 1, k)
for i ← 0 to n − 1 do
    if A[i] < key then
        D[c] ← A[i]
        c ← c + 1
    end
end
Return D
```

Algorithm 6: Initialize D with an Unsorted array in $O(n)$ time

It is clear that from the above pseudocode the running time of the algorithm is $O(n) = O(n + k \log n)$

ii USING SORTED ARRAY

```
Algorithm: INITIALIZE(A)
D ← newARRAY(k)
c ← 0
key ← QUICKSELECT(A, 0, n − 1, k)
for i ← 0 to n − 1 do
    if A[i] < key then
        D[c] ← A[i]
        c ← c + 1
    end
end
MERGE-SORT(D)
Return D
```

Algorithm 7: Initialize D with a Sorted array in $O(n + k \log k)$ time

It is clear that from the above pseudocode the running time of the algorithm is $O(n + k \log k) = O(n + k \log n)$

iii USING MAX-HEAP

```
Algorithm: INITIALIZE(A)
D ← newARRAY(k)
c ← 0
key ← QUICKSELECT(A, 0, n − 1, k)
for i ← 0 to n − 1 do
    if A[i] < key then
        D[c] ← A[i]
        c ← c + 1
    end
end
BUILD-MAXHEAP(D)
Return D
```

Algorithm 8: Initialize D with a Max-heap in $O(n + k)$ time

It is clear that from the above pseudocode the running time of the algorithm is $O(n + k) = O(n + k \log n)$

$\square$

(b) (3 points) For each of the choices (i)-(iii), compute the best running time for the TRAVERSE procedure you can think of. (In particular, tell your procedure.)

Solution:

TRAVERSE

i Apply merge-sort on D and return D. Time taken is $O(k \log k)$

ii The array is already sorted, so just return D. Time taken is $O(1)$

iii Keep performing Extract-Max from D and return in the reverse order. Time taken is $O(k \log k)$

□

(c) (5 points) For each of the choices (i)-(iii), compute the best running time for the INSERT procedure you can think of. (In particular, tell your procedure.)

Solution:

INSERT

i Find the maximum element in D. If the inserted element x is smaller than the maximum element, swap each other. Time taken is $O(k)$

ii Perform binary search in D. Insert x at the appropriate position and shift D to the right by one index from x and swap the last element of D with the position where x was inserted in A. Time taken is $O(k + \log k) = O(k)$

iii Find-Max in D. If the x is smaller than the max-element, Extract-Max and then Insert x into the heap D and insert the extracted maximum from the heap at the position where x was inserted in A. Time taken is $O(\log k)$

□