دانشکده‌ی علوم ریاضی

ساختمان داده مدرس: دکتر شهرام خزائی

تمرین سری دو

نام و نام‌خانوادگی : امیر ملک حسینی *شماره دانشجویی : ۴۰۱۱۰۰۵۲۸*

پرسش ۱

Akra-Bazzi:

$\Theta\left(n^{\rho}\left(1+\int_1^n \frac{f(x)}{x^{p+1}}dx\right)\right)$

and

$\sum_{i=1}^{k} a_i b_i^p = 1$

آ)

$\left(\frac{1}{2}\right)^p + \left(\frac{1}{2}\right)^p = 1 \longrightarrow p = 1$

$\Theta\left(n^{\rho}\left(1+\int_1^n \frac{x\log x}{x^{p+1}}dx\right)\right)$

$\Rightarrow T(n) = \theta(n\lg^2 n)$

$Wehave$ f(n) = nlog n and because we know from the Lemma 0.3 , 0.4 in the PDF that each $f1 = n$ and $f2 = \log n$ satisfy polynomial growth condition and because we know from the Lemma 0.1 in the PDF that if two function $f1$ and $f2$ satisfy polynomial growth condition then function $f3 = f1 * f2$ does it as well. so $f(n) = n\log n$ satisfies polynomial growth condition.

ب)

$2*\left(\frac{1}{2}\right)^p + \frac{8}{9}*\left(\frac{3}{4}\right)^p = 1 \Rightarrow p = 2$

$T(n) = \Theta(n^2(1+\int_1^n \frac{1}{x\log x}\frac{d}{dx}))$

In $n = 1$ the integral value is infinite , we can conclude that the running time of this part is constant and is O(1) so we can start the integeral bounds from $n = 2$. So the final value of integral is this:

⇒T(n) = $\theta(n^2 \lg(\lg n))$

Because we know from Lemma 0.3 in PDF that log(f(n)) satisfies polynomial growth condition and by Theorem 0.5 we know that function $f(n) = n^\alpha \log^n b \log^\gamma \log n$ satisfies the polynomial growth and $\log_a b = \frac{1}{\log_b a}$ then we can conclude that $f(n) = \frac{n^2}{\log n}$ also satisfies the condition.

پ)

$\frac{4}{3} * (\frac{1}{2})^p + \frac{16}{3} * (\frac{1}{4})^p = 1 \Rightarrow p = 2$

$T(n) = \Theta(n^2(1 + \int_1^n \frac{\log \log x}{x} \frac{d}{dx}))$

By Theorem 0.5 of the PDF we know that function $f(n) = n^\alpha \log^n b \log^\gamma \log n$ satisfies the polynomial growth so the function $f(n) = n^2 \log(\log n)$ also satisfies the condition.

ت)

Consider the integral of $\frac{1}{x}$ from 1 to $n+1$. This is equal to $\ln(n+1) - \ln(1) = \ln(n+1)$. Now, consider the sum of $\frac{1}{k}$ for k from 1 to n. This can be approximated by the integral by taking each term $\frac{1}{k}$ to be the height of a rectangle with width 1, and noticing that the sum of the areas of these rectangles is both more than the area under the curve $\frac{1}{x}$ from 1 to n (which is $\ln(n)$), and less than the area under the curve $\frac{1}{x}$ from 1 to $n+1$ (which is $\ln(n+1)$).
Therefore, we have $\ln(n) < \sum_{k=1}^{n} \frac{1}{k} < \ln(n+1)$.
Subtracting $\ln(n)$ from all parts of this inequality gives $0 < \sum_{k=1}^{n} \frac{1}{k} - \ln(n) < 1$.
As n goes to infinity, the difference $\sum_{k=1}^{n} \frac{1}{k} - \ln(n)$ approaches a constant, called β. Therefore, we can write $\sum_{k=1}^{n} \frac{1}{k} = \ln(n) + \beta + O\left(\frac{1}{n}\right)$, where $O\left(\frac{1}{n}\right)$ represents terms that go to 0 as n goes to infinity.
So, the sum of $\frac{1}{k}$ for k from 1 to n is $\ln(n)$ plus a constant

پرسش ۲

The algorithm is like this: We create two pointers i and j and a boolean value .The i pointer iterate over the context and j pointer keep track of 'X' place in each iteration.The boolean value checks that if we have multi 'X' or not, because we should only insert one 'X' if it happens,so when we find a match between word and a part of a context, we set boolean to true and if not, set it to false. This way we can only insert new 'X' only if the boolean value is false.For comparing word with context we iterate over context by i+=length(word) and check if it is a match or not. If yes the boolean will be true. j pointer job is just saving 'X' position and show us where should we insert the next 'X'.

This algorithm is in-place because we don't add any memory to the program and we just switch member's position in array.
Because we switch member's position in this problem, at the end we just output the part of array that we need.
The running time of this algorithm is O(m*n) if length(word) be m and length(context) be n.
Because in the program first we iterate over context , its running time is O(n) and then we iterate over the word to compare each chosen part of context with the word to find out if they match together or not , running time of this part is O(m) and since we do these two for loops connected, the overall running time would be O(m*n) .
This is the psuedocode:

```
function edit(context, word):
i = 0
j = 0
replaced = false
while i < length(context):
    if context[i:i+length(word)] == word:
        if replaced == false:
            context[j] = 'X'
            j = j + 1
            replaced = true
        i = i + length(word)
    else:
        context[j] = context[i]
        i = i + 1
        j = j + 1
        replaced = false
if replaced == true:
  j = j - 1
 return context[0:j]
```

پرسش ۳

In this problem we define two variables with initial value equal to zero "candidate" and "counter".
Then we iterate over the array and if counter is equal to zero we set the candidate value to ith member of the array and in each iteration we compare "candidate" value with ith member of the array and if they were equal, we increase "counter" value by one and if they were not, we decrease "counter" value by one.
We continue doing this to reach the end of the array and at the end if $length(array)/2 < counter$ it means that the answer is postive and the array is normal and if not, the

answer is negative.
The running time of this program is O(n) because we just iterate over the array once and order of its additional memory is O(1) because we added a constant memory to the program.

here is the psuedocode of this program:

```
function findMajorityElement (array):
    candidate = 0
    counter = 0
    for element in array:
        if counter == 0:
            candidate = element
            counter = 1
        else:
            if element == candidate:
                counter = counter + 1
            else:
                counter = counter - 1
    count = 0
    for element in array:
        if element == candidate:
            count = count + 1
    if count > length (array) / 2:
        return true
    else:
        return false
```

We can prove this algorithm with induction:

Suppose there is a majority element m in the input array, which means that it occurs more than half of the size of the array.
Let c be the element stored by the algorithm and i be the counter iterating on elements. We can show by induction on the number of elements processed that i is always non negative and that c is always equal to m whenever i is positive.
Base case: After processing the first element, i = 1 and c is equal to the first element. If the first element is m, then c = m and i is positive. If the first element is not m, then c is not equal to m but i is still non-negative.
Inductive step: Suppose after processing k elements, i is non negative and c is equal to m whenever i is positive. We consider two cases for the (k+1)th element x: If i =

0, then the algorithm sets c = x and i = 1. If x is m, then c = m and i is positive. If x is not m, then c is not equal to m but i is still non negative. If i > 0, then the algorithm compares x to c and either increases or decreases i. If x is m, then x = c and the algorithm increases i. This preserves the invariant that c = m and i is positive. If x is not m, then x != c and the algorithm decreases i. This preserves the invariant that i is non negative. Therefore, by induction, i is always non negative and c is always equal to m whenever i is positive.
Since m is the majority element, it occurs more than half of the size of the array. This means that the number of times the algorithm increases i is more than the number of times it decreases it. Therefore, i must be positive after processing all the elements. By the invariant, this implies that c is equal to m.
So, the algorithm correctly returns m as the majority element.

پرسش ۴

پرسش ۵

پرسش ۶

Algorithm of the problem works like this:
At the beginning of the program a Random number will be selected.Then there is a for loop in range length of the array.
When this loop begins, it sets a value to variable ”dest” and this value only depends to value of Random number that has been chosen beacuse i always starts from 1 so the first value of ”dest” always will be $1 + x$ and because the ’x’ value is chosen randomly between n numbers, so the probability of choosing a value for ’x’ is $1/n$.This means that the probability of allocating a number to ”dest” is $1/n$.
At the end of for loop A1 will be located in Bdest and because we showed that the probability of allocating a number to dest is $1/n$ then we can conclude that locating A1 in B has the same probability $1/n$.
When we execute the loop more that once we realize that in other executions of this for loop, the value of ”dest” increases by 1 each time and if it is more than length of A it will be decreased by n to insure that the new position is within the bounds of the array .This means that each time the algorithm will obey from a constant pattern that is been defined beforehand.
So the ”dest” value can only get one value in step two of iteration therefore A2 can only be placed in one position of B.
If the loop goes on we can see that for all A2 to An there is only one possible way. So all of their values only depends on the random number that has been chosen in first place.
Thus the probability of locating Ai in B is:
$(1/n) * 1 * 1... * 1 = 1/n$
Although the algorithm is not uniform because it does not generate all possible per-

mutations with equal probability. In a uniform random permutation algorithm, each possible permutation should be generated with equal probability. But, in the provided algorithm, the number of permutations it can generate is less than the total number of possible permutations.
For example, if we have an array with three members, the total number of possible permutations is $3! = 6$. But this algorithm can only generate three different permutations. Therefore, not all possible permutations are generated with equal probability, and therefore the algorithm is not uniform.In other word, some of permutations will be generated be $1/n$ probability and others will be generated with 0 probability so it is not uniform.