Gebze Technical University
Department of Computer Engineering
CSE 321 Introduction to Algorithm Design
Fall 2020
Midterm Exam (Take-Home)
November 25th 2020-November 29th 2020

Student ID and Name	Q1 (20)	Q2 (20)	Q3 (20)	Q4 (20)	Q5 (20)	Total
Dilara Karakas 171044010						

Read the instructions below carefully

- You need to submit your exam paper to Moodle by November 29th, 2020 at 23:55 pm as a single PDF file.

- You can submit your paper in any form you like. You may opt to use separate papers for your solutions. If this is the case, then you need to merge the exam paper I submitted and your solutions to a single PDF file such that the exam paper I have given appears first. Your Python codes should be in a separate file. Submit everything as a single zip file.

Q1. List the following functions according to their order of growth from the lowest to the highest. Prove the accuracy of your ordering. **(20 points)**

Note: Your analysis must be rigorous and precise. Merely stating the ordering without providing any mathematical analysis will not be graded!

a) 5^n

b) $\sqrt[4]{n}$

c) $\ln^3(n)$

d) $(n^2)!$

e) $(n!)^n$

Q2. Consider an array consisting of integers from 0 to n; however, one integer is absent. Binary representation is used for the array elements; that is, one operation is insufficient to access a particular integer and merely a particular bit of a particular array element can be accessed at any given time and this access can be done in constant time. Propose a linear time algorithm that finds the absent element of the array in this setting. Rigorously show your pseudocode and analysis together with explanations. Do not use actual code in your pseudocode but present your actual code as a separate Python program. **(20 points)**

Q3. Propose a sorting algorithm based on quicksort but this time improve its efficiency by using insertion sort where appropriate. Express your algorithm using pseudocode and analyze its expected running time. In addition, implement your algorithm using Python. **(20 points)**

Q4. Solve the following recurrence relations

a) $x_n = 7x_{n-1}-10x_{n-2}$, $x_0=2$, $x_1=3$ **(4 points)**

b) $x_n = 2x_{n-1}+x_{n-2}-2x_{n-3,}$ $x_0=2$, $x_1=1$, $x_2=4$ **(4 points)**

c) $x_n = x_{n-1}+2^n$, $x_0=5$ **(4 points)**

d) Suppose that a^n and b^n are both solutions to a recurrence relation of the form $x_n=\alpha x_{n-1}+\beta x_{n-2}$. Prove that for any constants c and d, ca^n+db^n is also a solution to the same recurrence relation. **(8 points)**

Q5. A group of people and a group of jobs is given as input. Any person can be assigned any job and a certain cost value is associated with this assignment, for instance depending on the duration of time that the pertinent person finishes the pertinent job. This cost hinges upon the person-job assignment. Propose a polynomial-time algorithm that assigns exactly one person to each job such that the maximum cost among the assignments (not the total cost!) is minimized. Describe your algorithm using pseudocode and implement it using Python. Analyze the best case, worst case, and average-case performance of the running time of your algorithm. **(20 points)**

CSE 321
INTRODUCTION TO ALGORITHM DESIGN
MIDTERM EXAM (TAKE-HOME)

DILARA KARAKAŞ
[illegible]

QUESTION 1:

$$5^n - \sqrt[4]{n} - \ln^3(n) - (n^2)! - (n!)^n$$

Take $\sqrt[4]{n}$ and $\ln^3(n)$

$$\lim_{n\to\infty} \frac{\sqrt[4]{n}}{\ln^3(n)} \overset{\text{L'Hospital}}{\Rightarrow} \lim_{n\to\infty} \frac{\frac{1}{4} n^{-3/4}}{3\ln^2(n)\frac{1}{n}} = \lim_{n\to\infty} \frac{\frac{1}{4} n^{\frac{+1}{4}}}{3\ln^2(n)} \xrightarrow{\text{L'Hospital}} \frac{\frac{1}{16} n^{\frac{-3}{4}}}{6\ln(n)\frac{1}{n}} =$$

$$\lim_{n\to\infty} \frac{\frac{1}{16} n^{\frac{1}{4}}}{6\ln(n)} \xrightarrow{\text{L'Hospital}} \lim_{n\to\infty} \frac{\frac{1}{64} n^{\frac{-3}{4}}}{6\frac{1}{n}} = \lim_{n\to\infty} \frac{\frac{1}{64} n^{\frac{1}{4}}}{6} = \infty //$$

Therefore $\boxed{\sqrt[4]{n} > \ln^3(n)}$

Take $\sqrt[4]{n}$ and 5^n

$$\lim_{n\to\infty} \frac{\sqrt{n}}{5^n} \to \lim_{n\to\infty} \frac{\log\sqrt[4]{n}}{\log 5^n} \Rightarrow \lim_{n\to\infty} \frac{\frac{1}{4}\log n}{n\log 5} = \frac{1}{\infty} = 0$$

Therefore, $\boxed{5^n > \sqrt[4]{n}}$

Take 5^n and $(n!)^n$

$$\log \to \begin{cases} \log 5^n \Rightarrow n\log 5 = n \\ \log(n!)^n \Rightarrow n\log(n!) \end{cases} \to \lim_{n\to\infty} \frac{n\log(n!)}{n} = \infty$$

Therefore, $\boxed{(n!)^n > 5^n}$

Take $(n^2)!$ and $(n!)^n$

$n! \sim \sqrt{2\pi n} \cdot \left(\frac{n}{e}\right)^n \Rightarrow$ Stirling Theorem

$(n^2)! \sim \sqrt{2\pi n^2} \cdot \left(\frac{n^2}{e}\right)^{(n^2)}$

$$\lim_{n\to\infty} \frac{(n!)^n}{(n^2)!} = \frac{\left(\sqrt{2\pi n} \cdot \left(\frac{n}{e}\right)^n\right)^n}{\sqrt{2\pi n^2}\left(\frac{n^2}{e}\right)^{(n^2)}} = \lim_{n\to\infty} \frac{\left(\sqrt{2\pi n}\right)^n \left(\left(\frac{n}{e}\right)^n\right)^n}{\sqrt{2\pi} \cdot n \cdot \left(\frac{n^2}{e}\right)^{(n^2)}}$$

$$= \lim_{n\to\infty} (2\pi)^{n-1} (\sqrt{n})^{n-2} \cdot \frac{1}{(n)^{(n^2)}}$$

$$\lim_{n\to\infty} \frac{(2\pi)^{n-2} \cdot \sqrt{2\pi} \cdot (\sqrt{n})^{n-2}}{(n)^{n-2} \cdot (n)^{n^2-n+2}} = \lim_{n\to\infty} \underbrace{\sqrt{2\pi}}_{\text{Constant}} \cdot \underbrace{\left(\frac{\sqrt{2\pi}}{\sqrt{n}}\right)^{n-2}}_{\frac{1}{\infty}=0} \cdot \underbrace{\frac{1}{(n)^{n^2-n+2}}}_{\frac{1}{\infty}=0}$$

$= \frac{1}{\infty} = 0$, Therefore $\boxed{(n^2)! > (n!)^n}$

$$\boxed{\ln^3(n) < \sqrt[4]{n} < 5^n < (n!)^n < (n^2)!}$$

QUESTION 2:

```
findMissingInteger ( L[0:n], R)
    if (L is empty
        return R
    endif
    leftArray
    rightArray
    for i in L then                          ⟶ n times
        if LSB is '0'
            add i to leftArray
        else
            add i to rightArray
        end if
    end for
    if size(leftArray) > size(rightArray) then
        add 1 to R
        FindMissingInteger (rightArray, R)  ⇒ T(n/2)
    else
        add 0 to R
        FindMissingInteger (leftArray, R)   ⇒ T(n/2)
    end if
end
```

LSB is last significant bits.

Each recursive call divides two to array. For loop is n times so

$T(n) = T(n/2) + n$ → Master Theorem.

$a = 1$, $b = 2$, $c = 1$ } if $a < b^c$ $\Rightarrow O(n^a)$

Complexity is $O(n)$

QUESTION 3:

```
precedure   QuickSort ( L[low:high], low, high)
        if high > low then
            pivot = partition (L, low, high)
            QuickSort (L, low, pivot-1)
            QuickSort (L, pivot+1, high)
            return L
        end if
end

precedure   partition ( L[low:high], low, high)
    pivot = arr[high]
    i, j = low
    for i in (low; high)
        if L[i] < pivot then
            a[i], a[j] = a[j], a[i]
            j++
        end if
    end for
    a[j], a[high] = a[high], a[j]
    return j.

precedure   InsertionSort (arr, low, n)
    for i in range(low+1, n+1)
        value = arr[i]
        j = i
        while j<low and arr[j-1]>val
            arr[j] = arr[j-1]
            j--
        end while
        arr[j] = value
    end for
end
```

```
procedure HybridSort (arr, low, high)
    while low < high
        if high - low < 9 then
            InsertionSort (arr, low, high)
            break
        else
            pivot = partition (arr, low, high)
            if pivot - low < high - pivot
                HybridSort (arr, low, pivot - 1)
                low = pivot + 1
            else
                HybridSort (arr, pivot + 1, high)
                high = pivot - 1
            end if
        end if
    end while
end
```

Quick sort is very fast with big data. However, it is slow with small data. So if we prefer to sort the insertion sort when an array with small data arrives, we'll improve it a bit. If the small data array is already sorted, there is a transition from $O(n^2)$ to $O(n)$.

The smaller of array size

Worst Case $O(n^2)$
Best Case $O(1)$
Average Case $O(n^2)$
Almost sorted case $O(1)$

The biggest of array size

Worst case $O(n^2)$
Best case $O(n \log n)$
Average case $O(n \log n)$
Almost Sorted case $O(n^2)$

QUESTION 4:

a) $x_n = 7x_{n-1} - 10x_{n-2}$, $x_0 = 2$, $x_1 = 3$

$r^2 - 7r + 10 = 0 \Rightarrow$ char eqv.

$r_1 = 5$, $r_2 = 2$ → char roots

$a_n = \alpha_1 (5)^n + \alpha_2 (+2)^n$

$x_0 \Rightarrow \alpha_1 + \alpha_2 = 2$

$5\alpha_1 + 2\alpha_2 = 3$

$3\alpha_1 = -1 \qquad \alpha_2 = 2 + 1/3$

$\alpha_1 = -1/3 \qquad \alpha_2 = 7/3$

$\Rightarrow x_n = -1/3 \cdot (5)^n + 7/3 \cdot 2^n$ //

b) $x_n = 2x_{n-1} + x_{n-2} - 2x_{n-3}$, $x_0 = 2$, $x_1 = 1$, $x_2 = 4$

$r^3 - 2r^2 - r + 2 = 0 \Rightarrow$ char eqv.

If $r = 1$

$$\begin{array}{r|l} r^3 - 2r^2 - r + 2 & r - 1 \\ \hline r^3 - r^2 & r^2 - r - 2 \\ \hline -r^2 - r + 2 & \\ -r^2 + r & \\ \hline -2r + 2 & \\ -2r + 2 & \\ \hline 0 & \end{array}$$

$(r-1)(r^2 - r - 2) \Rightarrow r_1 = 1, r_2 = 2, r_3 = -1$

↳ char roots

$\alpha_1 1^n + \alpha_2 2^n + \alpha_3 (-1)^n$

$x_0 \Rightarrow \alpha_1 + \alpha_2 + \alpha_3 = 2$

$x_1 \Rightarrow \alpha_1 + 2\alpha_2 - \alpha_3 = 1$

$x_2 \Rightarrow \alpha_1 + 4\alpha_2 + \alpha_3 = 4$

$2\alpha_1 + 3\alpha_2 = 3$

$2\alpha_1 + 6\alpha_2 = 5$

$\alpha_1 = 1/2 \qquad \alpha_2 = 2/3 \qquad \alpha_3 = 5/6$

Therefore, $x_n = 1/2\, 1^n + 2/3\, 2^n + 5/6\, (-1)^n$

c) $x_n = x_{n-1} + 2^n$, $x_0 = 5$

Homojen Part

$x = 1 \rightarrow$ first Root

$F(n) = 2^n$

$P(n) = c\,2^n$ → Put in equation

$\frac{c2^n}{2^{n-1}} = \frac{c2^{n-1}}{2^{n-1}} \cdot \frac{2^n}{2^{n-1}} \Rightarrow 2c = c + 2$

$c = 2$

$a(n) = \alpha_1 (1^n) + 2 \cdot 2^n$ → Heterojen part

$x_0 = 5 \Rightarrow \alpha_1 + 2 = 5$

$\alpha_1 = 3$

Therefore, $a(n) = 3(1)^n + 2^{n+1}$

$= 3 + 2^{n+1}$

d) $x(n) = \alpha x(n-1) + \beta x(n-2)$

$c/a^n = \alpha a^{n-1} + \beta a^{n-2}$

$d/b^n = \alpha b^{n-1} + \beta b^{n-2}$

$c\alpha^n = c\alpha a^{n-1} + c\beta a^{n-2}$

$+ \; d\beta^n = d\alpha b^{n-1} + d\beta b^{n-2}$

$c\alpha^n + d\beta^n = \alpha(ca^{n-1} + db^{n-1}) + \beta(ca^{n-2} + db^{n-2})$

$x(n) \qquad x(n-1) \qquad x(n-2)$

QUESTION 5:

First of all, I will talk a little bit about my algorithm. I created the person and job class to solve the problem. I randomly allocated and created the time that a person can do any job in the class. Then I made a distribution of work by putting myself as an employer. Let's think of it this way, we have a network job. We commissioned whoever did this as soon as possible. At the same time, I kept an object that keeps the status of doing business or not in the class, so that two jobs are not for one person. In this way, we were able to minimize the work.

Pseudocode: Assignment (people, jobs)

fillPeopleJobs (people, jobs) → O(n.M)

i=0

while i < length(people)

    index = findIndex (people, i) → O(n)

    people[index].setJob(jobs[i])

    i++

end while → O(n.M + n²) = O(n²)

end

n times

Pseudocode fillPeopleJobs (people, jobs)

```
i=0
while  i < length(people)  → n is people[] length
        j=0
        while j < length(job)  → m is job[] length
                people[i].add_jobs(jobs[i])  → O(1)
                j++
        end while
        i++
end while                    → O(n.m)
end
```

n. times

Pseudocode findIndex (people, index)

```
i=0
minValue
returnIndex
while  i < length(people)  → n is people[] length
        if people[i].hasJob == false
                if people[i].jobs[index] < minValue → O(1)
                        returnIndex = i
                        minValue = people[i].jobs[index]
                end if
        end if
        i++
end while                    → O(n)
return returnIndex
end
```

n times

→ The while loop of the findIndex function always returns n times. There is no case that could break this. So, $O(n)$

→ With some logic, the time complexity of the fillPeopleJobs function is $O(n^2)$

→ finaly, the while loop of the function we need to examine returns n times. But it calls findIndex function on every return. So $O(n^2)$ comes from that loop. The sum is $O(n^2+n^2) = O(n^2)$ with $O(n^2)$ coming from the fillPeopleJobs function called before loop.

So Time complexity = $O(n^2)$