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
171044065 Berke Belgin						

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) $(n!)^n$

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

For $\ln^3 n < \sqrt[4]{n}$:

$$\lim_{n\to\infty} \frac{ln^3 n}{\sqrt[4]{n}} \to Lopital \to$$

$$\lim_{n\to\infty} \frac{3.ln^2 n.(\frac{1}{n})}{\frac{1}{4}.n^{-\frac{3}{4}}} = \lim_{n\to\infty} 12.\frac{ln^2 n}{\sqrt[4]{n}} \to Lopital \to$$

$$\lim_{n\to\infty} 12.\frac{2.lnn.(\frac{1}{n})}{\frac{1}{4}.n^{-\frac{3}{4}}} = \lim_{n\to\infty} 96.\frac{lnn}{\sqrt[4]{n}} \to Lopital \to$$

$$\lim_{n\to\infty} 96.\frac{(\frac{1}{n})}{\frac{1}{4}.n^{-\frac{3}{4}}} = \lim_{n\to\infty} 384.\frac{1}{\sqrt[4]{n}} = 0, \text{ thus } \mathbf{\ln^3 n < \sqrt[4]{n}}$$

For $\sqrt[4]{n} < 5^n$:

$$\lim_{n\to\infty} \frac{\sqrt[4]{n}}{5^n} \to Lopital \to \lim_{n\to\infty} \frac{\frac{1}{4}.n^{-\frac{3}{4}}}{5^n.ln5} = \lim_{n\to\infty} \frac{1}{4.\sqrt[4]{n^3}.5^n.ln5} = 0, \text{ thus } \mathbf{\sqrt[4]{n} < 5^n}$$

For $5^n < (n!)^n$:

Since, for every $n > 2$, $(n!)^n > (n!)$ is true,

$$\lim_{n\to\infty} \frac{n!}{(n!)^n} = \lim_{n\to\infty} \frac{1}{(n!)^{n-1}} = 0,$$

$$\lim_{n\to\infty} \frac{5^n}{n!} = \lim_{n\to\infty} \frac{5.5.5.5}{1.2.3.4}.\coprod_{i=5}^{n} \frac{5}{i} = \lim_{n\to\infty} \frac{625}{24}.\coprod_{i=5}^{n} \frac{5}{i} = 0, \text{ thus } \mathbf{5^n < (n!)^n}$$

For $(n!)^n < (n^2)!$:

As Stirling's approximation states, $\ln n! = n.\ln n - n + O(\ln n)$

$\ln(n!)^n = n.\ln n = n.(n.\ln n - n + O(\ln n))$

$\ln(n^2!) = n^2\ln n^2 - n^2 + O(\ln n^2)$

$$\lim_{n\to\infty} \frac{\text{n.(n.lnn – n + O(lnn))}}{n^2 lnn^2 - n^2 + O(lnn^2)} = \lim_{n\to\infty} \frac{n^2.(\text{lnn} - 1 + \frac{\text{O(lnn)}}{\text{n}})}{n^4.(lnn^2 - 1 + \frac{O(lnn^2)}{n^2})} = \lim_{n\to\infty} \frac{1}{n^2} = 0, \text{ thus } \mathbf{(n!)^n < (n^2)!}$$

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)**

```
PROCEDURE findAbsent(bitArray, arraySize, bitSize)
    resultInt = []
    FOR bitIndex = bitSize - 1 TO -1 DO     → m times loop which is kind of a constant where m is bit size (8, 16, 32 etc.)
        evenCount = 0
        evenArray = []
        oddCount = 0
        oddArray = []
        FOR arrayIndex = 0 TO arraySize DO        → n times loop which is the amount of numbers in the array
            IF bitArray[arrayIndex * bitSize + bitIndex] == 0 THEN
                evenCount += 1
                FOR bit = 0 TO bitSize DO → m times loop again which is constant bit size
                    evenArray.append(bitArray[arrayIndex * bitSize + bit])
                ENDFOR
            ELSE IF bitArray[arrayIndex * bitSize + bitIndex] == 1 THEN
                oddCount += 1
                FOR bit = 0 TO bitSize DO → m times loop again which is constant bit size
                    oddArray.append(bitArray[arrayIndex * bitSize + bit])
                ENDFOR
            ELSE
                RETURN [-1]
            ENDIF
        ENDFOR
        IF evenCount <= oddCount THEN
            bitArray = evenArray
            arraySize = evenCount
            resultInt.insert(0, 0)
        ELSE
            bitArray = oddArray
            arraySize = oddCount
            resultInt.insert(0, 1)
        ENDIF
    ENDFOR
    RETURN resultInt
END
```

The above algorithms complexity is O(n) when we expect n amount of bits. When we assume n is the integer amount, then we can say that there will be $m^2.n$ operations. Since we accept m as a constant which is not an expanding value (8, 16, 32 etc.) for 32 bit integers, the actual complexity will be $32^2.n$ which is O(n) in Big O Notation. The above algorithm first checks every integers first bit then copies that number with its all bits into odd numbers array or even numbers array, if there are more 1s than 0s for the first bits, the algorithm traces the array with odd numbers again or vice versa. Then it checks all the remaining numbers second bits and makes the same operation again and so on. While tracing the numbers, it inserts the unbalanced (if even number count is greater it inserts 1) number into resulting number array. Then returns the result when every bit is checked.

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)**

PROCEDURE insertionSort(array, low, high) → **insertion sort is $O(n^2)$**

```
PROCEDURE insertionSort(array, low, high)
    FOR index = low + 1 TO high + 1 DO
            WHILE shiftIndex > low AND array[shiftIndex - 1] > element DO
                    array[shiftIndex] = array[shiftIndex - 1]
                    shiftIndex -= 1
            ENDWHILE
            array[shiftIndex] = element
    ENDFOR
END

PROCEDURE partition(array, low, high)
    pivot = array[high]
    j = low
    FOR i = low TO high DO
            IF array[i] < pivot THEN
                    temp = array[i]
                    array[i] = array[j]
                    array[j] = temp
                    j += 1
            ENDIF
    ENDFOR
    temp = array[j]
    array[j] = array[high]
    array[high] = temp

    RETURN j
END

PROCEDURE hybridQuickSort(array, low, high)
    WHILE low < high DO
            IF high - low + 1 < 10 THEN
                    insertionSort(array, low, high)
                    BREAK
            ELSE
                    pivot = partition(array, low, high)
                    IF pivot - low < high – pivot THEN
                            hybridQuickSort(array, low, pivot - 1)
                            low = pivot + 1
                     ELSE
                            hybridQuickSort(array, pivot + 1, high)
                            high = pivot-1
                    ENDIF
            ENDIF
    ENDWHILE
END
```

Despite Quick Sort is an effective algorithm for large arrays, it is not the most efficient algorithm for short array sizes so when we combine Insertions sorts performance for short arrays and quick sorts performance for larger arrays we can have one single algorithm which suits both conditions. Insertion limit is set to 10 in my code so the complexity of insertion sort becomes not n^2 but 10n. So, the general algorithm will be nlogn for greater parts and 10n for smaller chunks.

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)**

a)

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

$X_n = a^n$, $\frac{a^n}{a^{n-2}} = \frac{7.a^{n-1}}{a^{n-2}} - \frac{10.a^{n-2}}{a^{n-2}} \rightarrow a^2 = 7a - 10 \rightarrow a^2 - 7a + 10 = 0$

$a_1 = 5$, $a_2 = 2$,

$X_n = C_1.\ a_1^n + C_2.\ a_2^n = C_1.\ 5^n + C_2.\ 2^n$

$X_0 = 2 = C_1 + C_2$

$X_1 = 3 = 5C_1 + 2C_2$

$C_1 = \frac{-1}{3}$, $C_2 = \frac{7}{3}$

$X_n = \frac{-1}{3}.\ 5^n + \frac{7}{3}.\ 2^n$

b)

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

$X_n = a^n$, $\frac{a^n}{a^{n-3}} = \frac{2.a^{n-1}}{a^{n-3}} + \frac{a^{n-2}}{a^{n-3}} - \frac{2.a^{n-3}}{a^{n-3}} \rightarrow a^3 = 2a^2 + a - 2 \rightarrow a^3 - 2a^2 - a + 2 = 0$

$a_1 = 1$, $a_2 = 2$, $a_3 = -1$

$X_n = C_1.\ a_1^n + C_2.\ a_2^n + C_3.\ a_3^n = C_1.\ 1^n + C_2.\ 2^n + C_3.\ (-1)^n$

$X_0 = 2 = C_1 + C_2 + C_3$

$X_1 = 1 = C_1 + 2C_2 - C_3$

$X_2 = 4 = C_1 + 4C_2 + C_3$

$C_1 = \frac{1}{2}$, $C_2 = \frac{2}{3}$ $C_3 = \frac{5}{6}$

$X_n = \frac{1}{2}.\ 1^n + \frac{2}{3}.\ 2^n + \frac{5}{6}.\ (-1)^n$

c)

$X_n = x_{n-1} + 2^n$, $x_0 = 5$

$X_n^h = X_{n-1}$
$X_n = a^n$

$\frac{a^n}{a^{n-1}} = \frac{a^{n-1}}{a^{n-1}} \rightarrow$ a= 1

$X_n^h = C_1.1^n$
$X_n^p = C_2.2^n$

$X_n^p = x_{n-1}^p + 2^n \rightarrow$ $C_2.2^n = C_2.2^{n-1} + 2^n \rightarrow C_2 = \frac{c^2}{2} + 1$,
$C_2 = 2$

$X_n = C_1 + C_2.\ 2^n$, $X_0 = C_1 + C_2 = 5$, $C_1 = 3$
$X_n = 3 + 2.2^n$

d)

$X_n = \alpha.X_{n-1} + \beta.X_{n-2}$

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

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

So,

$\mathbf{c.a^n + d.b^n = c.\alpha.a^{n-1} + c.\beta.a^{n-2} + d.\alpha.b^{n-1} + d.\beta.b^{n-2}}$

is a solution.

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)**