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
171044046 Esra Eryılmaz						

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) $(n!)^n$

→ Compare $(n^2)!\,,\ (n!)^n$ → $\lim_{n\to\infty} \frac{(n^2)!}{(n!)^n}$ → $\boxed{\text{Using Stirling Formula } \; n! \cong \sqrt{2\pi n}\cdot\left(\frac{n}{e}\right)^n}$

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

$$= \lim_{n\to\infty} \frac{\sqrt{2\pi}\cdot n\cdot n^{n^2}}{\sqrt{2\pi}^n\cdot\sqrt{n}^n\cdot 1} = \lim_{n\to\infty} \frac{n\cdot n^{-n/2}\cdot n^{n^2}}{1} = \infty \qquad \text{so } \boxed{(n^2)! > (n!)^n}$$

constants

Scanned with CamSc

→ Compare $(5^n, \sqrt[4]{n})$ ⇒ $\lim_{n\to\infty} \frac{5^n}{\sqrt[4]{n}} = \lim_{n\to\infty} \frac{\overbrace{5^n}^{\text{Exponential}}}{\underbrace{n^{1/4}}_{\text{polynomial}}} = \infty$

2nd way

Also we can take logarithm for both functions:

$\log 5^n \to n.\log 5$

$\log \sqrt[4]{n} \to \frac{1}{4} \log n$

(Exponential grows faster than polynomial)

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

→ Compare $(5^n, (n!)^n)$ ⇒ $\lim_{n\to\infty} \frac{5^n}{(n!)^n}$ → $\boxed{\text{Using stirling Formula}}$

$= \lim_{n\to\infty} \frac{5^n}{(\sqrt{2\pi n}\cdot(\frac{n}{e})^n)^n} = \lim_{n\to\infty} \left(\frac{5}{\sqrt{2\pi n}\cdot(\frac{n}{e})^n}\right)^n = \lim_{n\to\infty} \left(\frac{\overbrace{5 . e^n}^{\text{constants}}}{\sqrt{2\pi n}\cdot \underbrace{n^n}_{\text{Exponential}\uparrow}}\right)^n = 0$

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

→ Compare $(\ln^3(n), 5^n)$ ⇒ $\lim_{n\to\infty} \frac{\ln^3 n}{5^n} = \frac{\infty}{\infty}$ $\boxed{\text{Using L'Hospital}}$

$= \lim_{n\to\infty} \frac{\frac{3.\ln^2 n}{n}}{\ln 5 . 5^n} = \lim_{n\to\infty} \frac{3.\ln^2 n}{n.\ln 5.\underbrace{5^n}_{\text{Exponential}\uparrow \text{ Grows fast}}} = 0$ So $\boxed{5^n > \ln^3 n}$

→ Compare $(\sqrt[4]{n}, \ln^3(n))$ ⇒ $\lim_{n\to\infty} \frac{\overbrace{\sqrt[4]{n}}^{\text{polynomial}}}{\underbrace{\ln^3 n}_{\text{Cubic grows faster than polynomial}}} = 0$

So $\boxed{\ln^3 n > \sqrt[4]{n}}$

Comparing all of my results:

$(n^2)! > (n!)^n$

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

$(n!)^n > 5^n$

$5^n > \ln^3 n$

$\ln^3 n > \sqrt[4]{n}$

$\underbrace{\sqrt[4]{n}}_{\text{polynomial}} < \underbrace{\ln^3(n)}_{\text{cubic}} < \underbrace{5^n}_{\text{Exponential}} < \underbrace{(n!)^n}_{\text{Exponential}} < \underbrace{(n^2)!}_{\text{Factorial}}$

$\boxed{b < c < a < e < d}$

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)**

Pseudocode :

```
procedure FindAbsentInteger ( L[], bits)
    if ( bits == 0)
        return 0

    left_partition[] ← values of L with MSB of 0
    right_partition[] ← values of L with MSB of 1

    if ( length (left_partition) ≤ length (right_partition))
            call FindAbsentInteger (left_partition , bits-1) << 1 | 0
                // MSB is 0
    else
            call FindAbsentInteger (right_partition, bits-1) << 1 | 1
                // MSB is 1
    end if
end

// while finding left_partition and right_partition, it access
// the array elements in constant time.
```

Explanation of the Algorithm with an example:

Suppose that the input array is : (doesn't matter sorted or not)

{0 , 1 , 2 , 3 , 4 , 5 , 7 , 8} 6 Missing

0000 0001 0010 0011 0100 0101 0111 1000

∝ Number of bits in the binary representation of a positive integer n is $= \log_2(n+1)$

∝ So if n is 8 then the numbers are represent using 4 bits. from 0000 to 1000.

➜ First partition the array using most significant Bit (MSB)

(0, 1, 2, 3, 4, 5, 7) – (8) continue with left_partition

➜ Partition the subarray using 2^{nd} MSB

(0, 1, 2, 3) – (4, 5, 7) continue with right_partition odd number of elements

➜ Partition the subarray using 3^{nd} MSB

(4, 5) – (7) continue with right_partition odd number of elements

➜ Partition the subarray using 4^{nd} MSB

(Nothing) – (7)

So we find the absent number → It is 6 (0110)

The first partition takes n operations

2^{nd} " " $\frac{n}{2}$ "

3^{nd} " " $\frac{n}{4}$ "

4^{nd} " " $\frac{n}{8}$ "

So the running time is $O(n + n/2 + n/4 + \cdots)$

Linear

$n + \frac{n}{2} + \frac{n}{4} + \cdots + 1 < 2n$ so it is $\underline{\underline{O(n)}}$

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)**

QuickSort algorithm average run time is $O(n\log n)$. But this algorithm can be as slow as $O(n^2)$ if we try to sort an array is already sorted. (if choosing pivot as a leftmost element)
But InsertionSort is more efficient than Quick Sort if we deal with small arrays. For small arrays, number of comparisons are less than Quick Sort.
As the Quick Sort works, it breaks our array into smaller and smaller arrays. So if the array become small enough then using Insertion Sort is more efficient than the Quick Sort.
We use InsertionSort when the size of the array is less than 10.

Pseudocode:

```
procedure QuickInsertionSort (L [low:high])
    while (high > low) do
        if (high-low < 10)          // This is where to use InsertionSort
            call InsertionSort (L [low: high])
            break

        else
            pivot = Partition (L [low :high])

            if ( pivot - low < high - pivot)
                call QuickInsertionSort ( L [low : pivot - 1])
                low = pivot + 1

            else
                call QuickInsertion Sort ( L [ pivot + 1 : high])
                high = pivot - 1

            end if
        end if
    end while
end
```

```
procedure Partition ( L[low : high])
    pivot = L[high]   // pick rightmost element
    i = J = low
    for i ← low to high do
        if( L[i] < pivot)
            swap L[i] and L[J]
            J = J+1
        end if
    end for
    swap L[J] and L[high]
    pivot = J
    return pivot
end
```

```
procedure InsertionSort ( L[low : n])
    for i ← low+1 to n do
        current = L[i]
        position = i
        while ((position > low) and
                (current < L[position-1])) do
            L[position] = L[position-1]
            position = position - 1
        end while
        L[position] = current
    end for
end
```

Expected running time:

∝ **Partition**

It performs high-low+2 operations
So runtime → $O(n)$

∝ **Insertion Sort**

General runtime analysis of insertion sort is that the outer loop inserts each element in turn ($O(n)$ iterations), and the inner loop moves that element to its correct place ($O(n)$ iterations), for total of $O(n^2)$.

But QuickInsertionSort() function leaves an array that can be sorted by size of at most threshold (approx. 10), each element moves at most threshold position. So the new analysis is $O(n * \text{threshold})$, which is equivalent to running insertion sort on each block separetely. which is linear as threshold is a constant.
→ $O(n)$

∝ **QuickInsertionSort**

In this function we call function recursively but inside the function we use insertion sort for small arrays (which iterates $O(n)$)
So average run time is $O(n\log n)$ but worst case is not $O(n^2)$ anymore. (with the effect of Insertion Sort) Much faster

Q4. Solve the following recurrence relations

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

b) $x_n = 2x_{n-1}+x_{n-2}-\cancel{3}\,\overset{2}{}x_{n-3}$, $x_0=2$, $x_1=1$, $x_2=4$ **(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) $X_n = 7X_{n-1} - 10X_{n-2}$ find roots: $r^2 - 7r + 10 = 0$ $\begin{matrix}-5\\-2\end{matrix}$ $\boxed{\begin{matrix}r_1=5\\r_2=2\end{matrix}}$

write characteristic eq.: $X_n = \alpha_1 r_1^n + \alpha_2 \cdot r_2^n$

$\boxed{X_n = \alpha_1 5^n + \alpha_2 2^n}$

using $x_0=2$, $x_1=3$:

$X_0 = \alpha_1 \cdot 5^0 + \alpha_2 \cdot 2^0$

$X_0 = \boxed{\alpha_1 + \alpha_2 = 2}$

$X_1 = \alpha_1 \cdot 5^1 + \alpha_2 \cdot 2^1$

$X_1 = \boxed{5\alpha_1 + 2\alpha_2 = 3}$

$-2\alpha_1 - 2\alpha_2 = -4$
$5\alpha_1 + 2\alpha_1 = 3$
$+$

$\alpha_1 = \frac{-1}{3}$ $\alpha_2 = \frac{7}{3}$

$X_n = \frac{-1}{3}5^n + \frac{7}{3}\cdot 2^n \Rightarrow \boxed{X_n = \frac{1}{3}\left(7\cdot 2^n - 5^n\right)} \rightarrow$ Result //

(b) $X_n = 2X_{n-1} + X_{n-2} - 3X_{n-3}$ find roots: $r^3 - 2r^2 - r + 2 = 0$

$(r-2)(r+1)\cdot(r-1) = 0$

$\boxed{r_1=2}$ $\boxed{r_2=-1}$ $\boxed{r_3=1}$

write characteristic eq.: $\boxed{X_n = \alpha_1 \cdot 2^n + \alpha_2 \cdot (-1)^n + \alpha_3 \cdot 1^n}$

using $X_0=2$, $X_1=1$, $X_2=4$:

$X_0 = \alpha_1 + \alpha_2 + \alpha_3 = 2$

$X_1 = 2\alpha_1 - \alpha_2 + \alpha_3 = 1$

$X_2 = 4\alpha_1 + \alpha_2 + \alpha_3 = 4$

$\rightarrow 3\alpha_1 + 2\alpha_3 = 3$

$\rightarrow 6\alpha_1 + 2\alpha_3 = 5$

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

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

$\boxed{X_n = \frac{2^{n+1}}{3} + \frac{5\cdot(-1)^n}{6} + \frac{1}{2}} \rightarrow$ Result

//

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

$$x_1 = x_0 + 2^1 \qquad x_1 - x_0 = 2^1$$
$$x_2 = x_1 + 2^2 \qquad x_2 - x_1 = 2^2$$
$$x_3 = x_2 + 2^3 \longrightarrow x_3 - x_2 = 2^3$$
$$\vdots \qquad \vdots$$
$$x_n = x_{n-1} + 2^n \qquad + \; x_n - x_{n-1} = 2^n$$

$$\boxed{x_n - x_0 = 2^1 + 2^2 + \dots + 2^n}$$

$$x_n - \overset{5}{x_0} = 2\left(1 + 2^1 + \dots + 2^{n-1}\right)$$

using geometric sum formula:

$$x_n - 5 = 2 \cdot \frac{(1-2^n)}{(1-2)} \Rightarrow x_n - 5 = \frac{2 - 2^{n+1}}{-1} \Rightarrow x_n = 2^{n+1} - 2 + 5$$

Result $\leftarrow$ $\boxed{x_n = 2^{n+1} + 3}$ //

---

d) Plugging a^n and b^n into recurrence relation:

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

Also plugging $c \cdot a^n + d \cdot b^n$ into the recurrence relation:

$$c \cdot a^n + d \cdot b^n = \alpha \cdot \left(c \cdot a^{n-1} + d \cdot b^{n-1}\right) + \beta\left(c \cdot a^{n-2} + d \cdot b^{n-2}\right)$$

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

$$0 = c\left(\underbrace{\alpha \cdot a^{n-1} + \beta \cdot a^{n-2}}_{a^n} - a^n\right) + d\left(\underbrace{\alpha \cdot b^{n-1} + \beta \cdot b^{n-2}}_{b^n} - b^n\right)$$

$$0 = c \cdot (a^n - a^n) + d\,(b^n - b^n)$$

$$0 = 0 \quad \checkmark$$

And it's prove that it holds

//

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)**