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
Muhammed ÖZKAN 151044084						

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) $(n!)^n$

Answer:

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

$$= \lim_{n\to\infty}\left(\frac{12ln^2(n)}{n^{\frac{1}{4}}}\right) = \lim_{n\to\infty}\left(\frac{24\,ln(n)\frac{1}{n}}{\frac{1}{4}n^{\frac{-3}{4}}}\right) = \lim_{n\to\infty}\left(\frac{96\,ln(n)}{n^{\frac{1}{4}}}\right)$$

$$= \lim_{n\to\infty}\left(\frac{96\frac{1}{n}}{\frac{1}{4}n^{\frac{-3}{4}}}\right) = \lim_{n\to\infty}\left(\frac{96*4}{n^{\frac{1}{4}}}\right) = \frac{\mathbf{1}}{\infty} = \mathbf{0} \rightarrow \boldsymbol{ln^3(n)} < \sqrt[4]{\boldsymbol{n}}$$

$$\lim_{n\to\infty}\left(\frac{\sqrt[4]{n}}{5^n}\right) = \lim_{n\to\infty}\left(\frac{\frac{1}{4}n^{\frac{-3}{4}}}{5^n ln5}\right) = \lim_{n\to\infty}\left(\frac{1}{4 * 5^n * ln5 * n^{\frac{3}{4}}}\right) = \frac{\mathbf{1}}{\infty} = \mathbf{0} \to \sqrt[4]{\boldsymbol{n}} < \mathbf{5}^{\boldsymbol{n}}$$

$$\lim_{n\to\infty}\left(\frac{5^n}{(n!)^n}\right) \quad n! = \sqrt{2\pi n}\left(\frac{n}{e}\right)^n \to \lim_{n\to\infty}\left(\frac{5^n}{\left(\sqrt{2\pi n}\left(\frac{n}{e}\right)^n\right)^n}\right) = \lim_{n\to\infty}\left(\frac{5}{\sqrt{2\pi n}\left(\frac{n}{e}\right)^n}\right)^n$$

$$= \left(\frac{\mathbf{1}}{\infty}\right)^{\infty} = \mathbf{0} \to \mathbf{5}^{\boldsymbol{n}} < (\boldsymbol{n}!)^{\boldsymbol{n}}$$

$$\lim_{n\to\infty}\left(\frac{(n!)^n}{(n^2)!}\right) \quad n! = \sqrt{2\pi n}\left(\frac{n}{e}\right)^n \to \lim_{n\to\infty}\left(\frac{\left(\sqrt{2\pi n}\left(\frac{n}{e}\right)^n\right)^n}{\sqrt{2\pi n^2}\left(\frac{n^2}{e}\right)^{n^2}}\right) = \lim_{n\to\infty}\left(\frac{n^{\frac{n}{2}}(2\pi)^{\frac{n}{2}}\left(\frac{n}{e}\right)^{n^2}}{n(2\pi)^{\frac{1}{2}}\left(\frac{n^2}{e}\right)^{n^2}}\right)$$

$$= \lim_{n\to\infty} n^{\left(\frac{n-2}{2}\right)}(2\pi)^{\left(\frac{n-1}{2}\right)}\left(\frac{1}{n}\right)^{n^2} = \mathbf{0} \to (\boldsymbol{n}!)^{\boldsymbol{n}} < (\boldsymbol{n}^{\mathbf{2}})!$$

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

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

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)**

Answer:

To solve this problem, it is necessary to talk about some features of the XOR operation.

Sequences of XOR operations

The XOR trick: If we have a sequence of XOR operations a ^ b ^ c ^ ..., then we can remove all pairs of duplicated values without affecting the result.

Commutativity allows us to re-order the applications of XOR so that the duplicated elements are next to each other. Since x ^ x = 0 and a ^ 0 = a, each pair of duplicated values has no effect on the outcome.

Because ^ is a bitwise operator, this will work regardless of what kind of values a, b and c are.

From the XOR trick we know that having a sequence of XOR statements means we can remove all duplicated arguments. If we just XOR all values in the given list, however, we cannot apply this trick because there are no duplicates:

A[0] ^ A[1] ^ ... ^ A[n]

What we can do additionally is to also XOR all values between 0 and n:

1 ^ 2 ^ ... ^ n ^ A[0] ^ A[1] ^ ... ^ A[n]

This will give us a sequence of XOR statements where elements appear as follows:

- All values in the given list now appear twice:
    - once from taking all the values between 0 and n
    - once because they were in the original list
- The missing value appears exactly once:
    - once from taking all the values between 0 and n

If we XOR all of this, we essentially remove all values that appear twice, thanks to the XOR trick. This means that we are left with the missing value, which happens to be exactly what we were looking for in the first place.

Now that we have expressed the required property, we can move on to the pseudocode part.

```
algorithm findmissingnumber(Arr)

         Result ← 0

         foreach K ∈  Arr do
             Result ← (Result XOR K)
         end for

         loop L from 0 to ((lenght of Arr) + 1) do
             Result ← (Result XOR L)
         end loop

  return Result
```

As can be seen in the pseudo code, it does XOR operation the binary array elements one by one with each other. Then it does XOR operation again for one more than the number of array elements. As a result of these operations, it finds the missing element.

If we calculate the time complexity over the pseudo code, there are 2 separate loops in our algorithm. Operations in the loop have constant time complexity. (O(1)). Both of these have O (n) time complexity. Consequently, the algorithm has a linear **O (n)** time complexity.

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)**

Answer:

Quicksort algorithm is efficient if the size of the input is very large. But, insertion sort is more efficient than quick sort in case of small arrays as the number of comparisons and swaps are less compared to quicksort. So we combine the two algorithms to sort efficiently using both approaches.

Insertion sort algorithm is used only when the size of the array is less than a threshold value(9 in this my algortihm).

The idea is to use recursion and continuously find the size of the array. If the size is greater than the threshold value(9), then the quicksort function is called for that portion of the array. Else, insertion sort is called.

```
algorithm insertion_sort(Arr, Low, N)
     loop I from Low+1 to N+1 do
          Val ← Arr[I]
          J ← I

          while (J greater than Low) and (Arr[J-1] greater than Val) do
                  Arr[J] ← A[J-1]
                  J ← J - 1
          end while

          Arr[J] ← Val

      end loop

 algorithm partition(Arr, Low, High)
        Pivot ← Arr[High]
        I ← Low
        J ← Low

        loop I from Low to High do
              if (Arr[I] less than Pivot) then
                    Temp ← Arr[I]
                    Arr[I] ← Arr[J]
                    Arr[J] ← Temp
                    J ← J + 1
              end if
        end loop

        Temp ← Arr[J]
        Arr[J] ← Arr[High]
        Arr[High] ← Temp

 return J
```

```
algorithm new_quick_sort(Arr, Low, High)
          while (Low less than High) do
                if ((High - Low + 1) less than 9) then
                      insertion_sort(Arr, Low, High)
                      break
                end if
                else
                      Pivot ← partition(Arr, Low, High)

                      if ((Pivot - Low) less than (High - Pivot)) then
                            new_quick_sort(Arr, Low, Pivot-1)
                            Low ← Pivot + 1
                      end if
                      else
                            new_quick_sort(Arr, Pivot+1, High)
                            High ← Pivot + 1
                      end else
                end else
 return Arr
```

The Quicksort is one of the fastest sorting algorithms for sorting large lists of data. The Insertion sort is a fast sorting algorithms for sorting very small lists that are already somewhat sorted. We combine these two algorithms to come up with a very simple and effective method for sorting large lists.

We start our sort by using a Quicksort. The Quicksort on average runs in O(n log(n)) time but can be as slow as O(n^2) if we try to sort a list that is already sorted and use the left most element as our pivot.

As the Quicksort works, it breaks down our list into smaller and smaller lists. Once the lists become small enough that an Insertion sort becomes more efficient then the Quicksort we switch to the Insertion sort to finish off the job.

We know that the best and average time complexes of quick sort is O (n log n). The best time complexity of the insertion sort is O (n). The time complexity is closer to O (n) when the insertion sort is applied in small-sized arrays.

In contrast, the time complexity in quick sort is O (n log n) regardless of the array size. The advantage of our algorithm is here. We use the insertion sort that uses O (n) time complexity instead of O (n log n) time complexity to sort sub-arrays of small sizes.

Analysis of NewQuickSort

By stopping when the sub-lists are of length k, the quicksort part becomes O(n log(n/k)) because the depth you have to go to is reduced by a factor of k.

The insertion sort would normally be O(n^2) because you move each element at most n/2 times on average, and the factor of 1/2 is ignored.

Because the insertion sort is now on a fixed length array, you only move each element at most k places. This results in the insertion sort part of your operation being O(nk).

In theory, larger values of k give faster sorts, but that relies on the number of items being sorted being large. In practise, the best value of k will depend on the value of n and can be found by differentiating the function in terms of k treating n as a constant.

As a result, the time complexity will run faster or slower than the threshold value we set in the algorithm.

All this shows us that the time complexity will be a value between **O (nk) and O (n log (n / k)).**

Still, the **O(n^2)** worst case is possible in extremely special cases.

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)**

Answer:

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

$\propto^2 = 7 \propto -10$ *characteristic equation*

$\propto^2 - 7 \propto +10 = 0$

$\propto \qquad -5 \qquad \propto_1 = 5$

$\propto \qquad -2 \qquad \propto_2 = 2$ *two distinct and real root*

$x(n) = c_1(\propto)^n + c_2(\propto)^n$

$x(n) = c_1(5)^n + c_2(2)^n$

$x(0) = c_1 + c_2 = 2$

$x(1) = 5c_1 + 2c_2 = 3$

$-2c_1 - 2c_2 = -4$

$5c_1 + 2c_2 = 3$

$3c_1 = -1 \qquad \boldsymbol{c_1 = -\frac{1}{3}} \qquad \boldsymbol{c_2 = \frac{7}{3}}$

$$\boldsymbol{x(n) = -\frac{1}{3}5^n + \frac{7}{3}2^n \qquad = \frac{1}{3}(7 * 2^n - 5^n)}$$

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

$\propto^3 = 2\propto^2 + \propto -2$ *characteristic equation*

$\propto^3 - 2\propto^2 - \propto + 2 = 0$

$(\propto -2)(\propto -1)(\propto +1) = 0$

$\propto_1 = -1, \propto_2 = 1, \propto_3 = 2$ *three distinct and real root*

$x(n) = c_1(\propto)^n + c_2(\propto)^n + c_3(\propto)^n$

$x(n) = c_1(-1)^n + c_2(1)^n + c_3(2)^n$

$x(0) = c_1 + c_2 + c_3 = 2$

$x(1) = -c_1 + c_2 + 2c_3 = 1$

$x(2) = c_1 + c_2 + 4c_3 = 4$

$\boldsymbol{c_1 = \frac{5}{6} \qquad c_2 = \frac{1}{2} \qquad c_3 = \frac{2}{3}}$

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

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

$$\left.\begin{aligned} x_0 &= 5 \\ x_1 &= x_0 + 2^1 = 5 + 2 = 7 \\ x_2 &= x_1 + 2^2 = 7 + 4 = 11 \\ x_3 &= x_2 + 2^3 = 11 + 8 = 19 \\ x_4 &= x_3 + 2^4 = 19 + 16 = 35 \\ x_5 &= x_4 + 2^5 = 35 + 32 = 67 \end{aligned}\right\} \textit{Guess } x_n = 3 + 2^{n+1}$$

proved if this guess is true

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

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

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

$$\boldsymbol{x(n) = 3 + 2^{n+1}}$$

d) **Lemma 1.** Let A and B be real numbers. A recurrence relation of the form

$$a_k = Aa_{k-1} + Ba_{k-2} \; for\; all\; integers\; k \geq 2, \qquad (1)$$

is satisfied by the sequence

$$1, t, t2, t3, \dots, tn, \dots,$$

where t is a nonzero real number, if, and only if, t satisfies the equation

$$t^2 - At - B = 0. \qquad (2)$$

Lemma 2. If $r_0, r_1, r_2, \dots$ and $s_0, s_1, s_2, \dots$ are sequences that satisfy the same second-order linear homogeneous recurrence relation with constant coefficients, and if C and D are any numbers, then the sequence $a_0, a_1, a_2, \dots$ defined by the formula

$$a_n = Cr^n + Ds^n \; for\; all\; integers\; n \geq 0$$

also satisfies the same recurrence relation.

Given a second-order linear homogeneous recurrence relation with constant coefficients, if the characteristic equation has two distinct roots, then Lemmas 1 and 2 can be used to find an explicit formula for any sequence that satisfies a second-order linear homogeneous recurrence relation with constant coefficients for which the characteristic equation has distinct roots, provided that the first two terms of the sequence are known. This is made precise in the next theorem.

Theorem 1 (Distinct Roots Theorem). Suppose a sequence $a_0, a_1, a_2, \dots$ satisfies a recurrence relation

$$a_k = Aa_{k-1} + Ba_{k-2} \; for\; all\; integers\; k \geq 2, \qquad (1)$$

for some real numbers A and B with $B \neq 0$. If the characteristic equation

$$t^2 - At - B = 0 \qquad (2)$$

has two distinct roots r and s, then $a_0, a_1, a_2, \dots$ is given by the explicit formula

$$a_n = Cr^n + Ds^n,$$

where C and D are the numbers whose values are determined by the values a_0 and a_1.

Remark. To say "C and D are determined by the values of a_0 and a_1" means that C and D are the solutions to the system of simultaneous equations

$$a_0 = Cr^0 + Ds^0 \; and\; a_1 = Cr^1 + Ds^1,$$

or, equivalently,

$$a_0 = C + D \; and\; a_1 = Cr + Ds.$$

This system always has a solution when $r \neq s$.

Proof. Suppose that for some real numbers A and B, a sequence $a_0, a_1, a_2, \dots$ satisfies the recurrence relation $a_k = Aa_{k-1} + Ba_{k-2}$, for all integers $k \geq 2$, and suppose the characteristic equation $t^2 - At - B = 0$ has two distinct roots r and s. We will show that

$$for\; all\; integers\; n \geq 0, a_n = Cr^n + Ds^n,$$

where C and D are numbers such that

$$a_0 = Cr^0 + Ds^0 \; and \; a_1 = Cr^1 + Ds^1 .$$

Let P(n) be the equation

$$a_n = Cr^n + Ds^n.$$

We use strong mathematical induction to prove that P(n) is true for all integers n ≥ 0. In the basis step, we prove that P(0) and P(1) are true. We do this because in the inductive step we need the equation to hold for n = 0 and n = 1 in order to prove that it holds for n = 2. **Show that P(0) and P(1) are true:** The truth of P(0) and P(1) is automatic because C and D are exactly those numbers that make the following equations true:

$$a_0 = Cr^0 + Ds^0 \; and \; a_1 = Cr^1 + Ds^1 .$$

Show that for all integers k ≥ 1, if P(i) is true for all integers i from 0 through k, then P(k + 1) is also true: Suppose that k ≥ 1 and for all integers i from 0 through k,

$$a_i = Cr_i + Ds_i .$$

We must show that P(k + 1):

$$a_{k+1} = Cr^{k+1} + Ds^{k+1} .$$

Now by the inductive hypothesis,

$$a_k = Cr^k + Ds^k \; and \; a_{k-1} = Cr^{k-1} + Ds^{k-1} ,$$

So

$$\begin{aligned} a_{k+1} &= Aa_k + Ba_{k-1} \\ &= A(Cr^k + Ds^k) + B(Cr^{k-1} + Ds^{k-1}) \\ &= C(Ar^k + Br^{k-1}) + D(As^k + Bs^{k-1}) \\ &= Cr^{k+1} + Ds^{k+1} . \end{aligned}$$

This is what was to be shown. [The reason the last equality follows from Lemma 1 is that since r and s satisfy the characteristic equation (2), the sequences r^0 , r^1 , r^2 , . . . and s^0 , s^1 , s^2 , . . . satisfy the recurrence relation (1).]

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)**