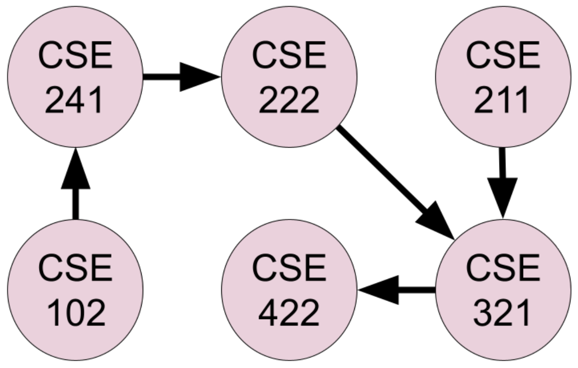
CSE 321 - Homework 3

Due date: 8/12/2022, 23:59

1. A directed acyclic graph (DAG) is a directed graph with no directed cycles. A DAG has at least one node with indegree (number of incoming edges) 0 and one node with outdegree (number of outgoing edges) 0.

   Topological sorting of a DAG is a linear ordering of nodes such that for every edge ab (an edge from node a to node b), a comes before b in the ordering. A DAG may have more than one topological order.

   Example: The following graph presents some of the courses in our department. The directed edges indicate the prerequisites. For instance, a student cannot enroll in $CSE321$ without successfully passing the courses $CSE211$ and $CSE222$. On the other hand, $CSE102$ and $CSE211$ do not have any prerequisites. Topological sortings of this graph should demonstrate the order of these courses such that if a student follows that ordering, they meet all of the prerequisite requirements.



   Such orderings are:

   - $CSE102 \rightarrow CSE241 \rightarrow CSE222 \rightarrow CSE211 \rightarrow CSE321 \rightarrow CSE422$
   - $CSE102 \rightarrow CSE241 \rightarrow CSE211 \rightarrow CSE222 \rightarrow CSE321 \rightarrow CSE422$
   - $CSE102 \rightarrow CSE211 \rightarrow CSE241 \rightarrow CSE222 \rightarrow CSE321 \rightarrow CSE422$
   - $CSE211 \rightarrow CSE102 \rightarrow CSE241 \rightarrow CSE222 \rightarrow CSE321 \rightarrow CSE422$

   (a) ***15 pts.*** Construct a DFS-based algorithm to obtain such an ordering for a given DAG.

   (b) ***15 pts.*** Construct a non-DFS-based algorithm to obtain such an ordering for a given DAG.

PS: The algorithms should find a single solution. You are not asked to find all possible orderings.

2. ***20 pts.*** Design an algorithm to calculate a^n, where $a \in Z$ and $n \in Z^+$, with a worst-case time complexity of $O(logn)$.

3. ***20 pts.*** Design an algorithm to solve $9x9$ sudoku game by using exhaustive search.

4. ***15 pts.*** Sort the following array in ascending order by using insertion sort, quick sort, and bubble sort algorithms. Determine whether they are stable sorting algorithms or not. Justify your answer on the given array.

   $array = \{6, 8, 9, 8, 3, 3, 12\}$

5. (a) ***5 pts.*** Give an explanation of the relation between brute force and exhaustive search.

   (b) ***5 pts.*** Learn about the basic idea of Caesar's Cipher and AES. Are they vulnerable to brute force attacks? Explain your answer.

   (c) ***5 pts.*** Why does the naive solution to primality testing (for input n, checking if $x \in \{2, 3, ..., n-1\}$ divides n) grow exponentially?

Important Notes

- For the first 3 problems, implement your solution in Python3. Write a driver function to test each of these algorithms (inputs should be given by the user). Gather all of the python code in a single .py file. Pay attention to clean coding.
- Write a report explaining the reasoning behind the algorithms you coded and analyze the worst-case time complexity of each of them. This report should also include your answers to Question 4 and Question 5. Write your report by using a program like MS Office and then convert it to a single PDF file. Pictures of handwritten works are **not accepted**.
- Upload two files only, a .py file and a .pdf file, **not a .zip or a .rar file**.

CSE 321 - Homework 3

Due date: 8/12/2022, 23:59

1. (a) The DFS-based solution consists of following steps:
    - Create the graph.
    - Use a list to keep the track of visitied vertices, initialize it with "False" values for each vertex.
    - Initialize an empty list to keep the output sequence.
    - Do the followings for each unvisited vertex.
      - Mark the vertex as visited.
      - Repeat this for all of the outgoing neighbors of this vertex.
      - Add the vertex to the output sequence.
    - Reverse the list (because it is generated in the reverse order).

   In this way, the algorithm visit all the vertices and creates a queue according to the neighborhood relations between them. If we assume there are v vertices and e edges, the algorithm will take $O(v + e)$ time to construct the sequence. Because it is analyzing each vertex and its outgoing neighbors (since outgoing neighbors are represented with edges, we can say that the algorithm is analyzing each vertex and each edge).

   (b) A solution without using DFS approach is to benefit in-degrees of vertices. The algorithm given in .py file follows these steps:
    - Create the graph.
    - Generate a list of in-degrees of vertices, let's call it D.
    - Generate a list of vertices with 0 in-degrees, let's call it Z.
    - Initialize an empty list to keep the output sequence.
    - Do the followings for each vertex from Z.
      - Add the vertex to the output sequence.
      - Delete the vertex from Z. Also, delete the edges that it is connected to from the edge list.
      - After the deletion of a vertex and some edges, update D.
      - According to D, update Z.

   Since a DAG has at least one vertex with 0 in-degree, this algorithm benefits this information. At each step, a vertex with 0 in-degree is added to the output sequence. Because if a vertex has 0 in-degree, it means that it has no prerequisites. And when a vertex is added to the output sequence, we should delete it from the graph and update the in-degree list. Because we already solved that vertex's position in the queue and we can ignore it along with the edges it is connected to. Similarly, if we have v vertices and e edges, this algorithm has $O(v + e)$ time complexity as the one above. Because it checks every vertex in a loop. Within the loop it checks all of the outgoing neighbors of the related vertex. The number of total outgoing neighbors are equal to the number of edges. Therefore the algorithm goes over v vertices and e edges.

2. Logarithmic time indicates solving the problem by dividing it into x pieces, where x is the base of the logarithm. In this question, base is 2. Therefore, we should calculate the half of the problem at each step. Since $a^n = a^{(n/2)^2}$, it is enough to calculate $a^{(n/2)}$ and multiply it with itself instead of calculating a^n. Again, instead of calculating $a^{(n/2)}$, we can calculate $a^{(n/4)}$. This will go up to a^1 which is a itself. In this way, we make $log_2 n$ calculations to compute a^n, thus the time complexity becomes $O(log_2 n)$.

   PS: If n is an odd number, we can do this operation for $a^{(n-1)}$ and then multiply the result with a since $a^n = a^{(n-1)} * a$.

3. The solution in the .py file follows these steps:

   - Generate a puzzle.
   - Do the followings for each cell, starting from the upper left one.
     - If the cell is out of the frame of the puzzle, this means that we have completed the solution, therefore return 'True'.
     - If the cell already has a value (the input values), continue with the next cell.
     - If the cell doesn't have a value, then try each value starting from 1 to 9, stop when found a valid solution for that cell. To check if a value is valid, do the followings.
       * Analyze the 3x3 box of that cell, check if it is valid in 3x3 box.
       * If it is valid in 3x3 box, then check if it is valid for 9x9 box.
       * If both are true, then assign that value to that cell.

   There are 9 possible solutions for each cell and we do these calculations for each cell. If we assume that the size of the puzzle is NxN, then we would have NxN cells, therefore the time complexity would be $O(9^{(NxN)})$. In this problem the time complexity is $O(9^{81})$.

4. In this question, 8 and 3 on the right side is emphasised with bold letters to visualize their position better. Also, the yellow colored part indicates the sorted part of the array.

   PS: Please contact me if you are colorblind and having an issue with understanding the solution, I can provide a different representation.

   (a) **Instertion Sort:**

Input:	6	8	9	**8**	3	**3**	12	
Step 1:	6	8	9	**8**	3	**3**	12	Starting with 6.
Step 2:	6	8	9	**8**	3	**3**	12	No change, since 6 < 8.
Step 3:	6	8	9	**8**	3	**3**	12	No change, since 8 < 9.
Step 4:	6	8	**8**	9	3	**3**	12	Swapped the **8** with 9, since **8** < 9.
Step 5:	3	6	8	**8**	9	**3**	12	Swapped 3 with 9, **8**, 8, and 6 respectively, since 3 is smaller than all of them.

Step 6:	3	**3**	6	8	**8**	9	12	Swapped **3** with 9, **8**, 8, and 6 respectively, since **3** is smaller than all of them.
Step 7:	3	**3**	6	8	**8**	9	12	No change, since 9 < 12.

Insertion sort is a stable sorting algorithm. As seen, the same numbers are sorted in the same order as in the input. The bold 3 was on the right side, and it is still on the right side after the sorting operation. Same goes for 8.

(b) **Quick Sort:** At each step, the element at the beginning of the unsorted part is selected as the pivot. Also, pivot is emphasised with italic letters and left & right indicators are emphasised with blue and red cells respectively.

Input:	6	8	9	**8**	3	**3**	12	
Step1:	*6*	8	9	**8**	3	**3**	12	6 is selected as the pivot. 8 is left and 12 is right.
Step2:	*6*	8	9	**8**	3	**3**	12	Since left>pivot and right>pivot we updated right.
Step3:	*6*	**3**	9	**8**	3	8	12	Swapped **3** with 8, since **3**<pivot and 8>pivot.
Step4:	*6*	8	9	**8**	3	**3**	12	Updated left.
Step5:	*6*	**3**	9	**8**	3	8	12	Since left>pivot and right>pivot we updated right.
Step6:	*6*	**3**	3	**8**	9	8	12	Swapped 3 with 9, since 3<pivot and 9>pivot.
Step7:	*6*	**3**	3	**8**	9	8	12	Updated left: left>pivot and right> pivot but we cannot update right. This is as far as we can go for pivot 6.
Step8:	3	**3**	*6*	**8**	9	8	12	Repositioned pivot by swapping it with 3.
Step9:	3	**3**	*6*	**8**	9	8	12	Implementing bubble sort on the left side of the pivot {3, 3}, but it is already sorted, no changes will be made.
Step10:	3	**3**	6	***8***	9	8	12	Implementing bubble sort on the right side of the pivot {**8**, 9, 8, 12}, ***8*** is selected as the pivot. 9 is left and 12 is right.
Step11:	3	**3**	6	***8***	9	8	12	Since left>pivot and right>pivot we updated right.
Step12:	3	**3**	6	***8***	8	9	12	We swapped 8 with 9, since 8=pivot and 9>pivot. But we cannot update left, this is as fas we can fo gor pivot ***8***.
Step13:	3	**3**	6	***8***	8	9	12	No reposition is made for pivot, since there is no smaller value than it in the list {8, 9, 12}.

As seen, in the end, the bold 8 is on the left side instead of the right side. Therefore, we can say that quick sort is not a stable sorting algorithm because it changes the order of the elements with same value.

(c) **Bubble Sort:** The bubble is emphasised with blue cells.,

Input:	6	8	9	**8**	3	**3**	12	
Step1:	6	8	9	**8**	3	**3**	12	No change, since 6<8.
Step2:	6	8	9	**8**	3	**3**	12	No change, since 8<9.
Step3:	6	8	**8**	9	3	**3**	12	Swapped **8** with 9, since 9>**8**.
Step4:	6	8	**8**	3	9	**3**	12	Swapped 3 with 9, since 9>3.
Step5:	6	8	**8**	3	9	**3**	12	Swapped **3** with 9, since 9>**3**.
Step6:	6	8	**8**	3	9	**3**	12	No change, since **3**<12.
Step7:	6	8	**8**	3	9	**3**	12	Last element is sorted, starting over.
Step8:	6	8	**8**	3	9	**3**	12	No change, since 6<8.
Step9:	6	8	**8**	3	9	**3**	12	No change, since 8=**8**.
Step10:	6	8	3	**8**	9	**3**	12	Swapped 3 with **8**, since **8**¿3.
Step11:	6	8	3	**8**	9	**3**	12	No change, since **8**<9.
Step12:	6	8	3	**8**	**3**	9	12	Swapped **3** with 9, since 9>**3**.
Step13:	6	8	3	**8**	**3**	9	12	Last two elements are sorted, starting over.
Step14:	6	8	3	**8**	**3**	9	12	No change, since 6<8.
Step15:	6	3	8	**8**	**3**	9	12	Swapped 3 with 8, since 8>3.
Step16:	6	3	8	**8**	**3**	9	12	No change, since 8=**8**.
Step17:	6	3	8	**3**	**8**	9	12	Swapped **3** with **8**, since **8**>**3**.

Step18:	6	3	8	**3**	**8**	9	12	Last three elements are sorted, starting over.
Step19:	3	6	8	**3**	**8**	9	12	Swapped 3 with 6, since 6>3.
Step20:	3	6	8	**3**	**8**	9	12	No change, since 6<8.
Step21:	3	6	**3**	8	**8**	9	12	Swapped **3** with 8, since 8>**3**.
Step22:	3	6	**3**	8	**8**	9	12	Last four elements are sorted, starting over.
Step23:	3	6	**3**	8	**8**	9	12	No change, since 3<6.
Step24:	3	**3**	6	8	**8**	9	12	Swapped **3** with 6, since 6>**3**.
Step25:	3	**3**	6	8	**8**	9	12	Last five elements are sorted, starting over.
Step26:	3	**3**	6	8	**8**	9	12	No change, since 3=**3**.

Bubble sort is a stable sorting algorithm. As seen, the same numbers are sorted in the same order as in the input. The bold 3 was on the right side, and it is still on the right side after the sorting operation. Same goes for 8.

5. (a) Exhaustive search is a brute force solution used for combinatorial problems while brute force is usually used for searching a string pattern. Both are time consuming, but exhaustive search is known to be relatively faster. Unlike brute force, exhaustive search has a direction to follow at each step.

   (b) Caesar's Cipher is vulnerable to brute force attack because the key may have 26 different values (for English alphabet). Trying out each of them will end up with the solution of the cipher text. On the other hand, AES is not vulnerable to brute force attacks. AES makes sure the key is as long as to make it impossible to crack in a meaningful amount of time. For example, cracking a 256 bit AES key would take $\approx 2.29 * 10^{32}$ years, because the key space is too large.

   (c) While computing the complexity, we should take the size of the input into consideration. For instance, if the input is an array, we should check the number of elements in the array. In this problem, n is not the size, it is just a value. The size of n is calculated in the number of bits that presents it in base 2. To store n in base 2, we need $log_2 n$ bits. Which makes the size of our input $n^{log_2 n}$ which is exponential, not polynomial.

   Why $n^{log_2 n}$? We check the divisibility of n with $[2, 3, ..., n-1]$. These numbers are represented as following:

   - 2 is represented with $\lceil log_2 2 \rceil = 1$ bit
   - 3 is represented with $\lceil log_2 3 \rceil = 2$ bits
   - 4 is represented with $\lceil log_2 4 \rceil = 2$ bits
   - ...
   - $n-1$ is represented with $\lceil log_2(n-1) \rceil$ bits.

As we see, we have $n - 2$ numbers and each of them needs at most $\lceil log_2(n-1) \rceil$ bits. While calculating the time complexity, we ignore constant values, therefore we see that we will make calculations for n numbers, each requiring at most $\lceil log_2 n \rceil$ bits. Thus, we will work on $n^{log_2 n}$ bits.