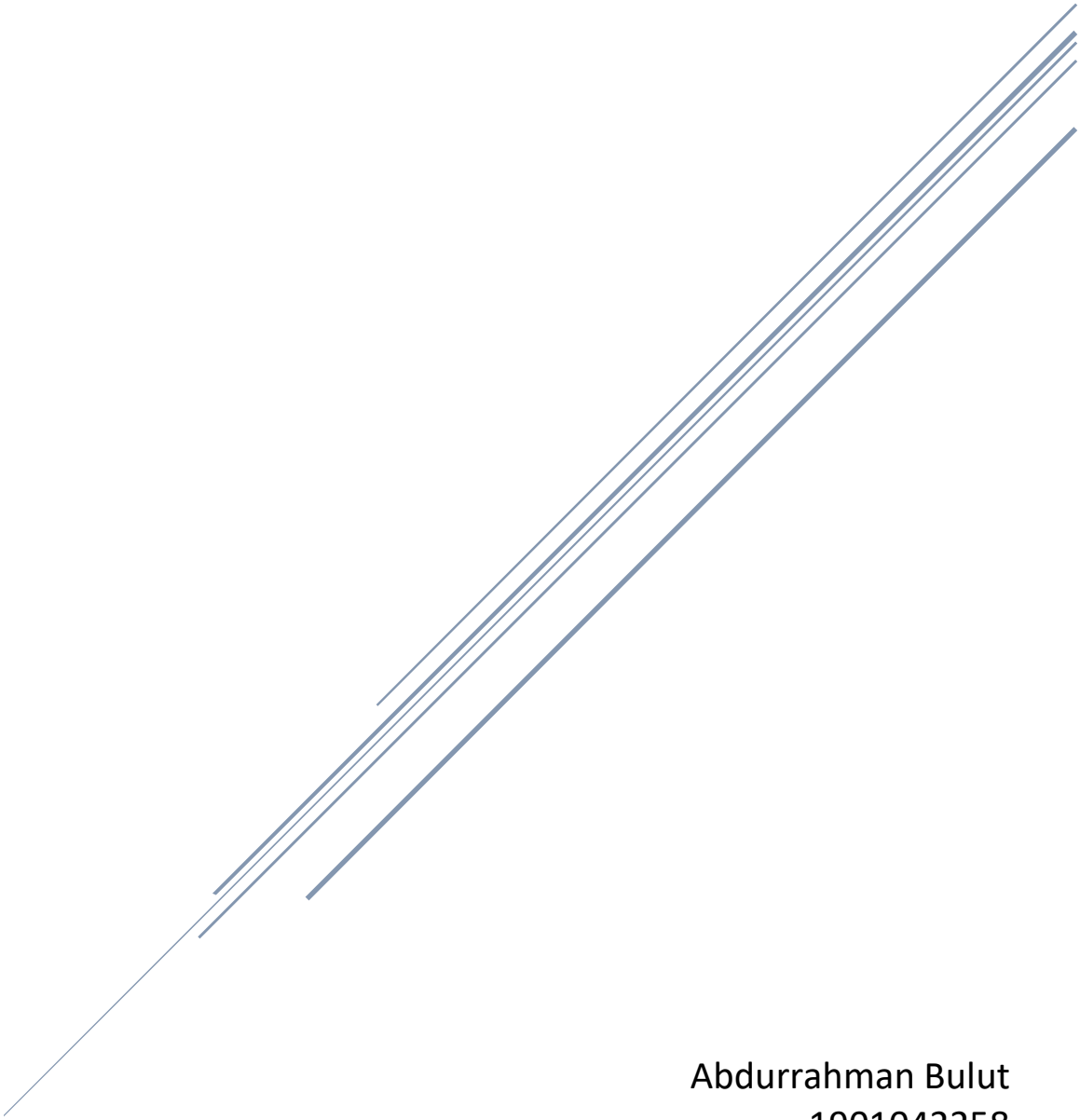
08/12/2022

CSE321

Introduction to Algorithm Design – Homework 3



Abdurrahman Bulut
1901042258

Question 1:

DFS-based algorithm:

Create a set of all nodes with course 0, and initialize an empty list of sorted nodes.

While the set of nodes with course 0 is non-empty:

Select a node with course 0 from the set.

Remove the node from the set and add it to the list of sorted nodes.

Decrement the course of each of the node's children by 1.

If a child's course becomes 0, add it to the set of nodes with course 0.

If the list of sorted nodes contains all of the nodes in the graph, return the list as the topological ordering. Otherwise, the graph contains a cycle and no topological ordering is possible.

The time complexity of a DFS-based algorithm for topological sorting on a DAG depends on the specific details of the algorithm, as well as the structure of the DAG itself. However, we can say that the time complexity of a DFS-based algorithm for topological sorting on a DAG with n nodes and m edges is O(n + m). The reason for this is that in a DFS-based algorithm for topological sorting, we typically visit each node in the DAG at least once, which takes O(n) time. Additionally, we also process each edge in the DAG at least once, which takes O(m) time. Therefore, the overall time complexity is O(n + m).

non-DFS-based algorithm:

Initialize a queue of nodes with indegree 0, and initialize an empty list of sorted nodes. While the queue of nodes with course 0 is non-empty:

Select a node with course 0 from the queue.

Remove the node from the queue and add it to the list of sorted nodes.

Decrement the course of each of the node's children by 1.

If a child's course becomes 0, add it to the queue of nodes with course 0.

If the list of sorted nodes contains all of the nodes in the graph, return the list as the topological ordering. Otherwise, the graph contains a cycle and no topological ordering is possible.

The time complexity of a non-DFS-based algorithm for topological sorting on a DAG depends on the specific details of the algorithm, as well as the structure of the DAG itself. However, in general, we can say that the time complexity of a non-DFS-based algorithm for topological sorting on a DAG with n nodes and m edges is also O(n + m). This is because any algorithm for topological sorting on a DAG must

visit each node in the DAG at least once, which takes O(n) time, and process each edge in the DAG at least once, which takes O(m) time. Therefore, the overall time complexity is O(n + m).

Question 2:

If n is even, it can be represented as (a^2)(n/2) and, if n is odd, as a * (a^2)((n-1)/2). We can calculate an in O(log n) time by continually dividing the exponent by 2 and carrying out the necessary calculations. It gives a recurrence relation.

```
3
4   def pow_func(a, n):
5       if n == 0:
6           return 1
7
8       if n % 2 == 0:
9           return pow_func(a * a, n / 2)
10      else:
11          return a * pow_func(a * a, (n - 1) / 2)
12
```

We can say recursive func in row 9 and 11 give us T(n/2) and total func is T(n) as a time complexity. First if (row 5) gives T(1) constant time. So,

T(n) = T(n/2) + 1

And this is a recurrence relation problem.

T(n/2) = T(n/4) + 1

T(n/4) = T(n/8) + 1

So,

T(n) = T(n/4) + 2

T(n) = T(n/8) + 3

.

.

T(n) = T(n/(2^k)) + k

Assume that, n/(2^k) = 1

k = logn

So, T(n) = T(1) + logn = O(logn)

Question 3: 9x9 sudoku with exhaustive search.

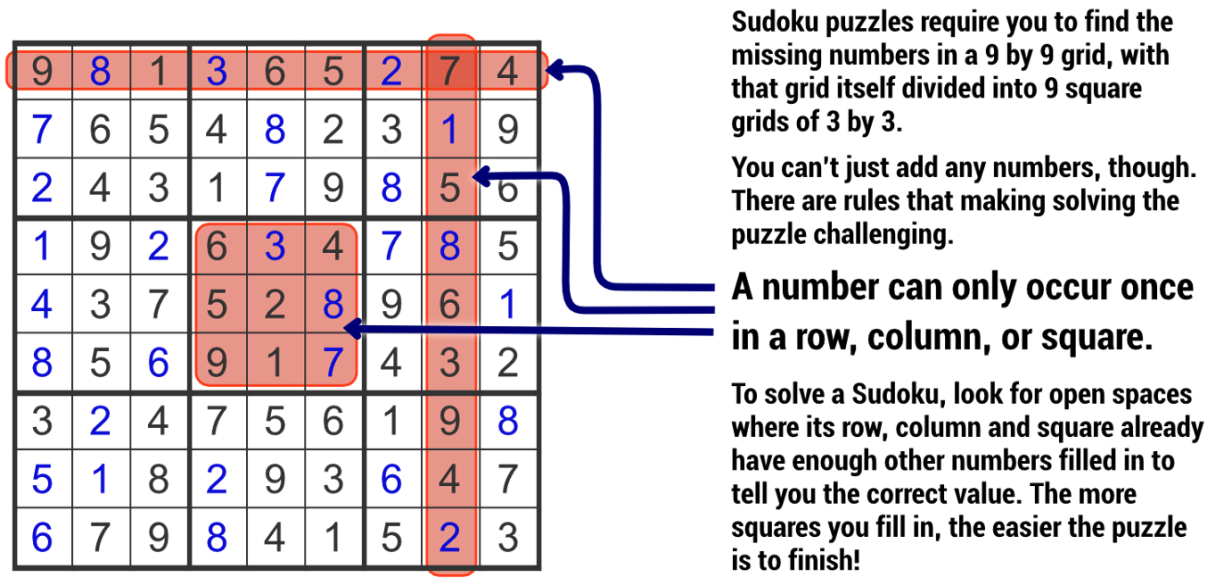


Figure from Medium/PasiduPerera

The algorithm that uses exhaustive search to solve a 9x9 Sudoku game:

1- Initialize an empty 9x9 grid to represent the Sudoku puzzle.

2- For each empty cell in the grid, try every possible integer from 1 to 9 as the value for that cell.

3- For each value that is tried, check if it is a valid solution by checking the following conditions:

4- Each row must contain all integers from 1 to 9.
Each column must contain all integers from 1 to 9.
Each 3x3 subgrid must contain all integers from 1 to 9.
If the value is a valid solution, then fill in the cell with that value and move on to the next empty cell.

5- If the value is not a valid solution, then discard it and try the next value.

6- Repeat steps 2-5 until all empty cells have been filled in with a valid value.

7- If a complete and valid solution is found, then return the solved Sudoku puzzle.

8- If no complete and valid solution is found, then return an error message indicating that the Sudoku puzzle is unsolvable.

By attempting every possible combination of values for each empty cell, this program searches exhaustively for a complete and legitimate solution to the Sudoku puzzle. The Sudoku puzzle has a temporal complexity of O(9^n), where n is the total number of vacant cells. This is because for each empty cell, the solve() function tries every number from 1 to 9 and recursively calls itself to solve the remaining empty cells. As a result, it will take longer to solve puzzles with more empty cells, but if a solution is possible, it will be found. It can even solve empty (filled with 0's) board.

Question 4:

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

Insertion Sort:

6 and 8 are sorted, move right

8 and 9 are sorted, move right

9 and 8 are not sorted, swap

New array = {6, 8, 8, 9, 3, 3, 12}

9 and 3 are not sorted, swap

New array = {6, 8, 8, 3, 9, 3, 12}

8 and 3 are not sorted, swap

New array = {6, 8, 3, 8, 9, 3, 12}

8 and 3 are not sorted, swap

New array = {6, 3, 8, 8, 9, 3, 12}

6 and 3 are not sorted, swap

New array = {3, 6, 8, 8, 9, 3, 12}

Continue index 5, array[5]

9 and 3 are not sorted, swap

New array = {3, 6, 8, 8, 3, 9, 12}

8 and 3 are not sorted, swap

New array = {3, 6, 8, 3, 8, 9, 12}

8 and 3 are not sorted, swap

New array = {3, 6, 3, 8, 8, 9, 12}

6 and 3 are not sorted, swap

New array = {3, 3, 6, 8, 8, 9, 12}

3 and 3 are sorted, continue

Continue index 6, array[6]

9 and 12 are sorted, Done

Result array = {3, 3, 6, 8, 8, 9, 12}

Insertion sort is a stable sorting algorithm, which means that it maintains the relative order of elements with the same value. In this case, the two 3s in the original array are preserved in their original order in the sorted array.

Quick Sort: array = {6, 8, 9, 8, 3, 3, 12}

Pivot is array[3] = 8

Move pivot to the end,

New array = {6, 8, 9, 12, 3, 3, 8}

i pointer for array[0], j pointer for array[5]

Move the i to the right until it reaches a value greater than or equal to the pivot.

i goes to the rigth, on array[1]

8 is equal to pivot, then

move j to the left until it crosses the left bound or finds a value less than the pivot.

j goes left, 3 is smaller then pivot, swap

New array = {6, 3, 9, 12, 3, 8, 8}

i goes to the right, on array[2]

9 is greater than pivot, then,

move j to left, on array[4]

3 is smaller then 9, swap

New array = {6, 3, 3, 12, 9, 8, 8}

i goes the right, on array[3]

12 is greater then pivot. i stops and j goes to the left

Bounds have crossed. Then, All elements to the right of the right bound are larger than or equal to the pivot, whereas all elements to the left of the left bound are less than the pivot.

move pivot to the final position

New array = {6, 3, 3, 8, 9, 8, 12}

Continue with left sublist

select pivot as array[1] = 3,

move pivot to the end,

New array = {6, 3, 3, 8, 9, 8, 12}

i pointer is on array[0] = 6, j pointer on array[1] = 3

i goes to the right until it reaches a value greater than or equal to the pivot.

6 is greater than pivot, the move j to the left

Bound crossed. It means that, all elements to the left of the left bound are less than the pivot and all elements to the right are greater than or equal to the pivot.

move pivot to the final position.

New array = {3, 3, 6, 8, 9, 8, 12}

Continue with left half array with right sublist, array[1]-array[2]

select pivot as array[1] = 3,

move pivot to the end,

New array = {3, 6, 3, 8, 9, 8, 12}

i pointer is on array[1] = 6, j pointer on array[1] = 6

i goes to the right until it reaches a value greater than or equal to the pivot.

6 is greater than pivot, the move j to the left

Bound crossed.

move pivot to the final position.

New array = {3, 3, 6, 8, 9, 8, 12}

Continue with left half array with right sublist, array[2]

It is single element.So, it is sorted already.

New array = {3, 3, 6, 8, 9, 8, 12}

Continue with right sublist

select pivot as array[5] = 8,

move pivot to the end,

New array = {3, 3, 6, 8, 9, 12, 8}

i pointer is on array[4] = 9, j pointer on array[5] = 12

i goes to the right until it reaches a value greater than or equal to the pivot.

9 is greater than pivot, the move j to the left

Bound crossed. It means that, all elements to the left of the left bound are less than the pivot and all elements to the right are greater than or equal to the pivot.

move pivot to the final position.

New array = {3, 3, 6, 8, 8, 12, 9}

3,3,6,8,8 are sorted now,

Continue with right half array with right sublist, array[5]-array[6]

select pivot as array[5] = 12,

move pivot to the end,

New array = {3, 3, 6, 8, 8, 9, 12}

partition the subarrays.

i pointer is on array[5] = 9, j pointer on array[6] = 12

i goes to the right until it reaches a value greater than or equal to the pivot.

move j to the left

Bound crossed.

move pivot to the final position.

New array = {3, 3, 6, 8, 8, 9, 12}

array[6] is on right position now.

Continue with right half array with right sublist, array[5]

It is single element.So, it is sorted already.

Final array = {3, 3, 6, 8, 8, 9, 12}

Quick sort is not a stable sorting algorithm, which means that it does not necessarily preserve the relative order of elements with the same value. In this case, the two 3s in the original array are not preserved in their original order in the sorted array.

Bubble Sort: array = {6, 8, 9, 8, 3, 3, 12}

we will move left to right swapping adjacent elements as needed. Each pass moves the next largest element into its final position.

Compare array[0] and array[1],

already sorted, move to the next

Compare array[1] and array[2],

already sorted, move to the next

Compare array[2] and array[3],

9 > 8, not sorted, then swap

New array = {6, 8, 8, 9, 3, 3, 12}

Compare array[3] and array[4],

9 > 3, not sorted, then swap

New array = {6, 8, 8, 3, 9, 3, 12}

Compare array[4] and array[5],

9 > 3, not sorted, then swap

New array = {6, 8, 8, 3, 3, 9, 12}

Compare array[5] and array[6],

already sorted.

First round is done. The last element is in its final position now.

Continue with second pass,

Compare array[0] and array[1],

already sorted, move to the next

Compare array[1] and array[2],

already sorted, move to the next

Compare array[2] and array[3],

8 > 3, not sorted, then swap

New array = {6, 8, 3, 8, 3, 9, 12}

Compare array[3] and array[4],

8 > 3, not sorted, then swap

New array = {6, 8, 3, 3, 8, 9, 12}

Compare array[4] and array[5],

already sorted. Second round is done. The index-5 element is in its final position now.

Continue with third pass,

Compare array[0] and array[1],

already sorted, move to the next

Compare array[1] and array[2],

8 > 3, not sorted, then swap

New array = {6, 3, 8, 3, 8, 9, 12}

Compare array[2] and array[3],

8 > 3, not sorted, then swap

New array = {6, 3, 3, 8, 8, 9, 12}

Compare array[3] and array[4],

already sorted. Third round is done. The index-4 element is in its final position now.

Continue with fourth pass,

Compare array[0] and array[1],

6 > 3, not sorted, then swap

New array = {3, 6, 3, 8, 8, 9, 12}

Compare array[1] and array[2],

6 > 3, not sorted, then swap

New array = {3, 3, 6, 8, 8, 9, 12}

Compare array[2] and array[3],

already sorted. Fourth round is done. The index-3 element is in its final position now.

Continue with fifth pass,

Compare array[0] and array[1],

already sorted, move to the next

Compare array[1] and array[2],

already sorted. Fifth round is done. The index-2 element is in its final position now.

Continue with sixth pass,

Compare array[0] and array[1],

already sorted. Sixth round is done. The index-1 element is in its final position now.

Done sorting!

Bubble sort is a stable sorting algorithm, which means that it maintains the relative order of elements with the same value. In this case, the two 3s in the original array are preserved in their original order in the sorted array.

Question 5:

a- The relation between brute force and exhaustive search.

Brute force and exhaustive search are two algorithms that can be used to solve problems by trying all possible combinations of solutions until a correct one is found.

The primary distinction between brute force and exhaustive search is the-fact that brute force is a method used when the issue is of finite size (pattern search). Exhaustive search, on the other hand, is a technique for solving large (permutational or combinational-related) problems. Furthermore, a thorough search takes less time than a forceful one.

Brute force algorithms are a type of exhaustive search that use a straightforward approach to solving a problem. They are often used when no other more efficient algorithms are known, or when the problem is small enough that the performance of the brute force algorithm is sufficient. Brute force algorithms are typically not very efficient and can have a high time complexity, but they are simple to implement and can provide a correct solution.

Exhaustive search is a more general class of algorithms that systematically try all possible combinations of solutions until a correct one is found. Exhaustive search algorithms can use more sophisticated techniques than brute force algorithms to find solutions, but they still have the same general approach of trying all possible combinations of solutions. Exhaustive search algorithms can be more efficient than brute force algorithms, but they can still be very time-consuming for large problems.

b- Caesar's Cipher and AES – Brute force attack

The Caesar cipher involves replacing each letter of the alphabet with the letter standing three places further down the alphabet. For example,

plain: meet me after the toga party
cipher: PHHW PH DIWHU WKH WRJD SDUWB

The alphabet is wrapped around, so that the letter following Z is A. We can define the transformation by listing all possibilities, as follows:

plain: a b c d e f g h i j k l m n o p q r s t u v w x y z
cipher: D E F G H I J K L M N O P Q R S T U V W X Y Z A B C

If it is known that a given ciphertext is a Caesar cipher, then a brute-force cryptanalysis is easily performed. because there are only 25 potential keys. Simply try all the 25 possible keys and get the plain text.

AES uses a 128-bit block size and a key size of 128, 192, or 256 bits and uses the same three key size alternatives but limits the block length to 128 bits. AES is a more sophisticated encryption algorithm that uses a combination of substitution and permutation operations to encrypt and decrypt messages. A number of AES parameters depend on the key length. Both Caesar's Cipher and AES are vulnerable to brute force attacks, where an attacker tries all possible combinations of keys or shift values until the correct one is found. For Caesar's Cipher, this would involve trying all possible shift values until the original message can be reconstructed from the ciphertext. For AES, this would involve trying all possible keys until the original message can be reconstructed from the ciphertext. Both Caesar's Cipher and AES are vulnerable to brute force attacks, where an attacker tries all possible combinations of keys or shift values until the correct one is found. For Caesar's Cipher, this would involve trying all possible shift values until the original message can be reconstructed from the ciphertext. For AES, this would involve trying all possible keys until the original message can be reconstructed from the ciphertext.

c- The naive solution to primality testing

For many cryptographic algorithms, it is necessary to select one or more very large prime numbers at random. Thus we are faced with the task of determining whether a given large number is prime. The naive solution to primality testing grows exponentially because the number of possible values of x grows exponentially with the size of the input n. More specifically, for a given input n, there are n - 1 possible values of x that need to be checked in order to determine if n is prime. This means that, for larger values of n, the number of possible values of x grows exponentially, which makes the algorithm grow exponentially in terms of time complexity. For example, if n = 100, then there are 99 possible values of x that need to be

checked in order to determine if n is prime. If n = 1000, then there are 999 possible values of x that need to be checked. And if n = 10000, then there are 9999 possible values of x that need to be checked. As the value of n grows, the number of possible values of x grows exponentially, which makes the naive solution to primality testing grow exponentially as well.