```
            ++currentDay;
        }
    }

    return count;
}
```

Day 7 / 100 :

Topic - Strings, Array

1 Problem statement: [Largest number](#) (medium)

Given a list of non-negative integers nums, arrange them such that they form the largest number and return it.

Since the result may be very large, so you need to return a string instead of an integer.

Example 1:
Input: nums = [10,2]
Output: "210"

Example 2:
Input: nums = [3,30,34,5,9]
Output: "9534330"

Solutions :

Approach 1 - Comparator Based Sorting

```
class Solution {
public:
    string largestNumber(vector<int>& nums) {
        vector<string> arr;
        for(auto i:nums){
          arr.push_back(to_string(i));
```

```
        }
        sort(begin(arr),end(arr),
        [](string &s1, string &s2){return s1+s2>s2+s1;});
        string res;
        for(auto s:arr)
          res+=s;
        while(res[0]=='0' && res.length()>1)
          res.erase(0,1);
        return res;
    }
};
```

but I want to explain this part of the code

- sort(begin(arr), end(arr), [](string &s1, string &s2){ return s1+s2>s2+s1; });

In these type of sorting, we sort the elements as per our choice by passing third argument in sort function.
For example:

- In sort(begin(arr), end(arr), [](string &s1, string &s2){ return s1+s2>s2+s1; });
- [](string &s1, string &s2){ return s1+s2>s2+s1; }) this part has meaning that:
- Binary function that accepts two elements in the range as arguments, and returns a value convertible to bool. The value returned indicates whether the element passed as first argument is considered to go before the second in the specific strict weak ordering it defines.
- The function shall not modify any of its arguments.
- Given two numbers X and Y, decide which number to put first – we compare two numbers XY (Y appended at the end of X) and YX (X appended at the end of Y). If XY is larger, then X should come before Y in output, else Y should come before. For example, let X and Y be 542 and 60. To compare X and Y, we compare 54260 and 60542. Since 60542 is greater than 54260, we put Y first.
- if we have written the condition [](string &s1, string &s2){ return s1+s2<s2+s1; }) then our function has arranged the numbers of array in smallest number.
- You can refer to http://www.cplusplus.com/reference/algorithm/sort/

Solutions :

Approach 2

```
class Solution {
public:
    string largestNumber(vector<int> &num) {
        sort(num.begin(), num.end(), [](int a, int b){
            return to_string(a)+to_string(b) >
to_string(b)+to_string(a);
        });
        string ans;
        for(int i=0; i<num.size(); i++){
            ans += to_string(num[i]);
        }
        return ans[0]=='0' ? "0" : ans;
    }
};
```

1. The code aims to arrange a list of non-negative integers to form the largest possible number.
2. It uses the sort function to sort the numbers in descending order based on a custom comparison function.
3. The comparison function concatenates the string representations of two numbers in two different ways and compares them in reverse order.
4. This sorting order ensures that the resulting array maximizes the formed number. For example, "330" is greater than "303", so 3 should come before 30 in the sorted array.
5. After sorting, the numbers are converted to strings and concatenated in the order they appear.
6. If the resulting string starts with '0', it means the largest number formed is 0, so the code returns "0" as the final result.
7. Otherwise, it returns the concatenated string, representing the largest number.

The key idea is to compare and sort the numbers based on their concatenated string representations to achieve the desired ordering.

2 Problem statement: Sudoku Solver (Hard)

Write a program to solve a Sudoku puzzle by filling the empty cells.

A sudoku solution must satisfy all of the following rules:

Each of the digits 1-9 must occur exactly once in each row.
Each of the digits 1-9 must occur exactly once in each column.
Each of the digits 1-9 must occur exactly once in each of the 9 3x3 sub-boxes of the grid.
The '.' character indicates empty cells.

Example 1:

Input: board =
[["5","3",".",".","7",".",".",".",".","."],
["6",".",".","1","9","5",".",".",".","."],
[".","9","8",".",".",".",".",".","6","."],
["8",".",".",".",".","6",".",".",".","3"],
["4",".",".","8",".","3",".",".",".","1"],
["7",".",".",".",".","2",".",".",".","6"],
[".","6",".",".",".",".",".","2","8","."],
[".",".",".",".","4","1","9",".",".","5"],
[".",".",".",".",".","8",".",".","7","9"]]
Output:
[["5","3","4","6","7","8","9","1","2"],
["6","7","2","1","9","5","3","4","8"],
["1","9","8","3","4","2","5","6","7"],
["8","5","9","7","6","1","4","2","3"],
["4","2","6","8","5","3","7","9","1"],
["7","1","3","9","2","4","8","5","6"],
["9","6","1","5","3","7","2","8","4"],
["2","8","7","4","1","9","6","3","5"],
["3","4","5","2","8","6","1","7","9"]]
Explanation: The input board is shown above, and the only valid solution is shown below:

5	3	4	6	7	8	9	1	2
6	7	2	1	9	5	3	4	8
1	9	8	3	4	2	5	6	7
8	5	9	7	6	1	4	2	3
4	2	6	8	5	3	7	9	1
7	1	3	9	2	4	8	5	6
9	6	1	5	3	7	2	8	4
2	8	7	4	1	9	6	3	5
3	4	5	2	8	6	1	7	9

Solutions :

Approach 1 - Brute Force

The brute force approach involves trying out all possible combinations of numbers until a valid solution is found. We can solve the Sudoku puzzle by backtracking and trying out all possible numbers for each empty cell.

```
bool isValid(vector<vector<char>>& board, int row, int col, char num)
{
    // Check if the number already exists in the same row
    for (int i = 0; i < 9; i++) {
        if (board[row][i] == num) {
            return false;
        }
    }

    // Check if the number already exists in the same column
    for (int i = 0; i < 9; i++) {
        if (board[i][col] == num) {
            return false;
        }
    }

    // Check if the number already exists in the same 3x3 box
    int startRow = 3 * (row / 3);
    int startCol = 3 * (col / 3);
    for (int i = 0; i < 3; i++) {
        for (int j = 0; j < 3; j++) {
            if (board[startRow + i][startCol + j] == num) {
                return false;
            }
        }
    }

    return true;
}

bool solveSudoku(vector<vector<char>>& board) {
    for (int row = 0; row < 9; row++) {
```

```
        for (int col = 0; col < 9; col++) {
            if (board[row][col] == '.') {
                for (char num = '1'; num <= '9'; num++) {
                    if (isValid(board, row, col, num)) {
                        board[row][col] = num; // Try out the number

                        if (solveSudoku(board)) {
                            return true; // Valid solution found
                        } else {
                            board[row][col] = '.'; // Undo the choice
and try the next number
                        }
                    }
                }
                return false; // No valid number found, backtrack
            }
        }
    }
    return true; // All cells filled, solution found
}
```

Explanation:

1. The isValid function checks whether it is valid to place the number num at the given row and col.
2. The solveSudoku function uses a nested loop to iterate over each cell in the Sudoku board.
3. If the current cell is empty (denoted by '.'), it tries out all numbers from '1' to '9' and checks if it is valid to place that number in the current cell.
4. If a number is valid, it places that number in the cell and recursively calls the solveSudoku function to solve the remaining empty cells.
5. If the solveSudoku function returns true, it means a valid solution has been found, and the recursion unwinds to return true.
6. If no valid number is found for the current cell, it returns false to backtrack to the previous cell and try the next number.
7. Finally, in the main function, we initialize the Sudoku board and call the solveSudoku function. If a solution is found, it prints the solved board; otherwise, it prints "No solution exists."

Solutions :

Approach 2 - Backtracking

```
bool isValid(vector<vector<char>>& board, int row, int col, char num)
{
    for (int i = 0; i < 9; i++) {
        if (board[row][i] == num) {
            return false;
        }
        if (board[i][col] == num) {
            return false;
        }
        if (board[3 * (row / 3) + i / 3][3 * (col / 3) + i % 3] ==
num) {
            return false;
        }
    }
    return true;
}

bool solveSudoku(vector<vector<char>>& board) {
    for (int row = 0; row < 9; row++) {
        for (int col = 0; col < 9; col++) {
            if (board[row][col] == '.') {
                for (char num = '1'; num <= '9'; num++) {
                    if (isValid(board, row, col, num)) {
                        board[row][col] = num; // Try out the number

                        if (solveSudoku(board)) {
                            return true; // Valid solution found
                        } else {
                            board[row][col] = '.'; // Undo the choice
and try the next number
                        }
                    }
                }
                return false; // No valid number found, backtrack
            }
        }
```

```
    }
    return true; // All cells filled, solution found
}
```

Explanation:

1. The efficient approach is similar to the brute force approach, but it introduces the concept of pruning by making small modifications to the isValid function.
2. In the isValid function, instead of using nested loops to check for validity, we use a single loop from 0 to 8 to check the entire row, column, and 3x3 box at once.
3. The row and column checks remain the same, but the 3x3 box check is modified. It calculates the starting row and column index of the current 3x3 box based on the given row and col.
4. The efficient approach reduces the number of iterations required to check for validity, making it more efficient than the brute force approach.
5. The rest of the code remains the same as the brute force approach.
6. Both the brute force and efficient approaches solve the Sudoku puzzle by backtracking and trying out all possible combinations of numbers. The efficient approach improves the efficiency by pruning and reducing unnecessary iterations.