Problem 6-1. Just a warm-up: Solve the following recursion:

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

Solution. If we think about the tree for this recurrence we see that in the first level we have n, in the second level we have $n/2 + n/4 = 3n/4$, and in the i-th level we have $(\frac{3}{4})^{i-1}n$ until we hit the base case on one of the branches. After that, each level will only have less than $(\frac{3}{4})^{i-1}n$.

So we can uppper bound $T(n)$ by:

$$\begin{aligned} T(n) &\leq \sum_{i=0}^{\infty} (\frac{3}{4})^i n \\ &= 4n \end{aligned}$$

From the first level alone, we have $T(n) \geq n$.

So we conclude $T(n) = \Theta(n)$.

Problem 6-2. Asymptotic analysis: Order the following asymptotic runtimes, from fastest to slowest (all logarithms are with base e, i.e. $e^{\log n} = n$):

$$2^{2^n} \qquad n! \qquad n2^n \qquad 10n \qquad \log(n!) \qquad n^{\log\log n} \qquad \binom{n}{4} \qquad e^n \qquad \left(\frac{\log n}{\log\log n}\right)^{\frac{\log n}{\log\log n}}$$

Solution. The order is:

$$\left(\frac{\log n}{\log\log n}\right)^{\frac{\log n}{\log\log n}} \qquad 10n \qquad \log(n!) \qquad \binom{n}{4} \qquad n^{\log\log n} \qquad n2^n \qquad e^n \qquad n! \qquad 2^{2^n}$$

Problem 6-3. In this exercise we will study the *Heaviest Non-Successive Numbers* problem:

Given a sequence $X = (x_1, x_2, \ldots, x_n)$ of positive integers, the index set $I \subseteq \{1, \ldots, n\}$ is called *legal*, if I does not contain successive indices from the sequence. For any index set J, we define its weight: $w(J) = \sum_{j \in J} x_j$. Here we are seeking for a fast algorithm to compute a legal index set of maximum weight.

For example, if $X = (12, 66, 23, 6)$, then $I = \{1, 3\}$ would be *legal* and $w(I) = 12 + 23 = 35$, whereas $I = \{1, 3, 4\}$ would not be legal because indices 3 and 4 are successive. The best solution is the set $I^* = \{2, 4\}$ with $w(I^*) = 66 + 6 = 72$.

1. We consider the following natural greedy algorithm:

- Start with the empty set $I = \varnothing$ and $J = \{1, \ldots, n\}$.
- While $J \neq \varnothing$, find index $j \in J$ with the maximum weight x_j, add x_j to I and delete indices $j, j-1, j+1$ from J.
- Output I.

Give a simple counterexample where the above greedy algorithm fails to find the optimum solution.

2. True or False: if I is the output of the greedy algorithm and I^* is a legal index set of maximum weight, then: $w(I) \geq \frac{1}{2}w(I^*)$
3. Give a dynamic programming algorithm that runs in time $O(n)$ and finds a legal set of maximum weight.

Solution.

1. $(1, 1+\varepsilon, 1)$. The greedy algorithm will pick the $1+\varepsilon$ element, whereas the optimum solution is 2.
2. True: If the greedy algorithm picks element j, then by the greedy criterion we have $x_j \geq x_{j-1}$ and $x_j \geq x_{j+1}$. Using induction, it follows that the greedy algorithm is a 2-approximation.
3. We use dynamic programming: Define $X_i = \{x_1, x_2, \ldots, x_i\}$ and let $S(X_i)$ to be the weight of the optimum solution for the sequence X_i. Suppose we knew $S(X_{n-1})$ and $S(X_{n-2})$ and we are focusing on whether to choose x_n or not. We get the recursion: $S(X_n) = max\{S(X_{n-1}), S(X_{n-2}) + x_n)\}$. Generally: $S(X_i) = max\{S(X_{i-1}), S(X_{i-2}) + x_i)\}$ for $i \geq 2$, $S(X_1) = x_1$ and $S(X_0) = 0$.

Problem 6-4. Subset Sum: Given a set of non-negative integers, and a value sum S, determine if there is a subset of the given set with sum equal to the given sum S.

```
Example: set = {3, 34, 4, 12, 5, 2}, sum = 9
Output:  True  //There is a subset (4, 5) with sum 9.
```

Solution. The basic idea is to observe that (n is the number of elements in the set and || means logic or):

```
isSubsetSum(set, n, sum) = isSubsetSum(set, n-1, sum) ||
                           isSubsetSum(set, n-1, sum-set[n-1])
Base Cases:
isSubsetSum(set, n, sum) = false, if sum > 0 and n == 0
isSubsetSum(set, n, sum) = true, if sum == 0
```

A naive implementation in Python:

```
# A recursive solution for subset sum
# problem

# Returns true if there is a subset
# of set[] with sum equal to given sum
def isSubSum(set,n, sum) :

    # Base Cases
    if (sum == 0) :
        return True
    if (n == 0 and sum != 0) :
        return False

    # If last element is greater than
    # sum, then ignore it
    if (set[n - 1] > sum) :
        return isSubSum(set, n - 1, sum);

    # else, check if sum can be obtained
    # by any of the following
    # (a) including the last element
    # (b) excluding the last element
    return isSubSum(set, n-1, sum) or isSubSum(set, n-1, sum-set[n-1])


# Driver program to test above function
set = [3, 34, 4, 12, 5, 2]
sum = 9
n = len(set)
if (isSubSum(set, n, sum) == True) :
    print("Found a subset with given sum")
else :
    print("No subset with given sum")
```

The above implementation is naive because similar to the Fibonacci computation from class, we may end up computing the same quantities many times from scratch. We can avoid this by using Dynamic Programming. We create a boolean 2D table $subset[][]$ and fill it in a bottom up manner. The value of $subset[i][j]$ will be true if there is a subset of $set[0..i-1]$ with sum equal to j, otherwise false. Finally, we return $subset[n][S]$.

```
# A Dynamic Programming solution for
# subset sum problem
# Returns true if there is a subset
# of set[] with sum equal to given sum
def isSubsetSum(st, n, sm) :

    # The value of subset[i][j] will be
    # true if there is a subset of
    # set[0..i-1] with sum equal to j
    subset=[[True] * (sm+1)] * (n+1)

    # If sum is 0, then answer is true
    for i in range(0, n+1) :
        subset[i][0] = True

    # If sum is not 0 and set is empty,
    # then answer is false
    for i in range(1, sm + 1) :
        subset[0][i] = False

    # Fill the subset table in bottom
    # up manner
    for i in range(1, n+1) :
        for j in range(1, sm+1) :
            if(j < st[i-1]) :
                subset[i][j] = subset[i-1][j]
            if (j >= st[i-1]) :
                subset[i][j] = subset[i-1][j] or subset[i - 1][j-st[i-1]]

    return subset[n][sm];

# Driver program to test above function
st = [1, 2, 3]
sm = 7
n = len(st)
if (isSubsetSum(st, n, sm) == True) :
    print("Found a subset with given sum")
else :
    print("No subset with given sum")
```

Problem 6-5. Let $a_1, a_2, \ldots, a_n$ be a sequence of n different integer numbers. The *swap-number* for this sequence is the number of pairs (a_i, a_j) with $i < j$ and $a_i > a_j$, i.e. it is the number of out-of-order pairs. For example, for the sequence 6,5,4,3,2,1 the *swap-number* is

15 (all pairs are out of order), for the sequence 5,1,2,6,3,4 the *swap-number* is 6 (the pairs are: (5,1),(5,2),(5,3),(5,4),(6,3),(6,4)) and finally for the sorted sequence 1,2,3,4,5,6 the *swap-number* is 0.

1. Compute the *swap-number* of the sequence $a_1, a_2, \ldots, a_n$ by providing an algorithm that runs in $O(n^2)$ time.
2. Compute the *swap-number* of the sequence $a_1, a_2, \ldots, a_n$ by providing a *divide and conquer* algorithm that runs in time $O(n \log n)$. Feel free to use as a black-box any subroutines we have seen in class.

Solution.

1. For every number we check how many numbers on the right are smaller than it is. We can do that in $O(n)$ time for each number hence $O(n^2)$ in total.

```
Merge&Count($A, B$)
    $i \leftarrow 1;\ j \leftarrow 1;\ k \leftarrow 1;\ r_{AB} \leftarrow 0;$
    while $i \leq |A|$ and $j \leq |B|$ do
        if $A[i] < B[j]$ then
            $C[k] \leftarrow A[i];\ k \leftarrow k+1;\ i \leftarrow i+1;$
        else
            $C[k] \leftarrow B[i];\ k \leftarrow k+1;\ j \leftarrow j+1;\ r_{AB} \leftarrow r_{AB} + |A| - i + 1;$
```

Figure 1: The main change inside the subroutine in mergesort.

2. The idea is based on Mergesort with a slight change in the part where we merge the subarrays. Suppose we split the initial sequence in two pieces A, B and we can recursively compute the swap-number r_A of A and the swap-number r_B of B. We can also assume that the two pieces are sorted. Then the swap number for the initial sequence r is just $r = r_A + r_B + r_{AB}$ where r_{AB} is number of out-of-order pairs of the form (a, b) with $a \in A, b \in B$. But we can compute this number while we are merging the two subarrays: Every time an element b from B is selected to be merged and go on top of the merged array we construct for A, B, the counter r_{AB} is incremented by the remaining number of elements in A that have not yet been merged, since all of the are larger than b. See Figure 1.

Problem 6-6. Here we study a variation of a sorting problem:

We want to make a construction and we have a box with n metal screws and a box with n metal nuts. All the screws have different size and all the nuts have different size. Every screw matches with exactly one nut. We are seeking to match all screws with their unique corresponding nut. There is however a constraint: We cannot compare the sizes of two screws neither the sizes of two nuts. We can only compare the size of one screw with the size of one nut and figure out if the screw is bigger, smaller or matches exactly with the nut (and of course similarly figure out if the nut is bigger, smaller or matches exactly with the screw).

Provide a randomised algorithm that runs in expected $O(n \log n)$ time and that will match all the screws with their unique corresponding nut.

Solution. Here the idea is based on Quicksort. We cannot of course use Quicksort by itself because of the pivot rule. Here is the main idea: We are going to pick a random nut and this will be used as a pivot for the partition of the screws, we are going to find in linear time the screw that corresponds to the chosen nut, and then this screw will be used as a pivot for the partition of the nuts. This partition procedure takes $O(n)$ time and we now have two smaller subsets of screws and corresponding nuts. We perform the above procedure recursively and the running time analysis is exactly the same as randomised Quicksort.