General rules: In solving these questions you may consult your book; You can discuss high level ideas with each other, but each student must find and write his/her own solution. You should upload the pdf file (either typed, or a clear scan) of your solution to mycourses.

1. (15 points) Consider the closest pair of points problem discussed in the class (Section 5.4), but now suppose that the distance between two points \((x_i, y_i)\) and \((x_j, y_j)\) is measured as \(|x_i - x_j| + |y_i - y_j|\) instead of the Euclidean distance. Modify the algorithm discussed in the class to solve this problem.

Solution: We can divide points into left half and right half based on their \(x\) coordinates. Suppose we have two pairs of closest points from the left half and the right half. We need to consider the fact that there might be a shorter distance with one end from the left half and another end from the right half. To handle this, we only need to consider points within \(\delta\) of line \(L\), where \(L\) is the line that separates the left half and the right half and \(\delta\) is the minimum distance that we have found so far (See Section 5.4 of the textbook for a proof).

We want a box to contain only 1 point because that would make the analysis easy. We can’t divide the \(2\delta\) strip into \(\frac{\delta}{2} \times \frac{\delta}{2}\) boxes, because a box could contain 2 points under our new metric. We could use \(\frac{\delta}{3} \times \frac{\delta}{3}\) boxes instead. We can sort all points in the strip by their \(y\) coordinates, and store in \(S_y\). We know that points which are 3 rows apart must be at least \(\delta\) apart. Therefore, for each point, we can check its distance with the next and previous 23 points in \(S_y\). We will update \(\delta\) if we find a smaller one. This completes the ”combining” step. Rest of the algorithm is the same.

2. (15 points) Consider an \(n \times n\) table where some cells are marked as forbidden. A frog is placed in the \((1, 1)\)-cell (that is lowest and left most cell). At every step, the frog can jump from its current cell \((i, j)\) to any cell \((i', j')\) that satisfies \(i' + j' = i + j + 1\) provided that the cell is not forbidden. We assume that \((1, 1)\) and \((n, n)\) are never forbidden.

Design and analyze an efficient algorithm that given the coordinates of the forbidden cells, finds the number of different paths that the frog can take to move from \((1, 1)\) to \((n, n)\).

Solution I: For \(k = 2, \ldots, 2n\), let \(d_k\) be the number of non-forbidden cells \((i, j)\) with \(i + j = k\). Note that the number of possible choices for the \((k - 1)\)-th cell on the path from \((1, 1)\) to \((n, n)\) is given by \(d_k\). Hence the total number of ways to get from \((1, 1)\) to \((n, n)\) is \(d_2 \times d_3 \times \ldots \times d_{2n}\). Assuming multiplication requires only \(O(1)\), the running time of this algorithm is \(O(n)\).

Solution II: For \(i, j\), define \(N[i, j]\) to be the number different ways that the frog can take to go from \((1, 1)\) to \((i, j)\). Set \(F[i, j] = 1\) if \((i, j)\) is forbidden and \(F[i, j] = 0\) otherwise. We have

\[
N[1, 1] = 1,
\]

and for \(i, j\) with \(i + j > 2\), we have \(N[i, j] = 0\) if \((i, j)\) is forbidden, and otherwise

\[
N[i, j] = \sum_{i', j': i' + j' = i + j - 1, \ F[i', j'] = 0} N[i', j'],
\]

1
which can also be written as

\[ N[i, j] = \sum_{i', j': i'+j' = i+j-1} N[i', j'] \times F[i', j'] = 0. \]

**Algorithm 1 Dynamic solution to problem 2**

1: Initialize \( N[i, j] := 0 \) for all \( i, j \).
2: for \( k = 0, \ldots, n \) do
3: for \( i = 1, \ldots, n - k \) do
4: Set \( j := i + k \)
5: if \( F[i, j] == 1 \) then
6: for \( i' = 1, \ldots, i+j-2 \) do
7: Set \( j' := i+j-1-i' \)
8: \( N[i, j] += N[i', j'] \times F[i', j'] \)

Each for loop runs for \( O(n) \), and thus the running time is \( O(n^3) \).

3. (15 points) Design and analyze a \( O(n^3) \) algorithm that given a sequence of \( n \) distinct numbers, finds the largest subsequence, such that each number is between the previous two numbers in the subsequence. In other words if \( a, b, c \) are three consecutive numbers in the subsequence, then \( \min(a, b) < c < \max(a, b) \). For example if the input is 1, 3, 9, 8, 5, 7, 6, then the output is 1, 9, 5, 7, 6

**Solution:** For \( 1 \leq i < j \leq n \), let \( D[i, j] \) be the length of the longest subsequence that has the above properties and furthermore ends at \( a_i, a_j \). Note that

\[ D[i, j] = 1 + \max\{D[t, i] : t < i, \min(a_t, a_i) < a_j < \max(a_t, a_i)\} \]

where we consider the maximum as 0 if no \( t \) satisfies those conditions. This can be used to write a simple dynamic program for the problem

**Algorithm 2 Solution to problem 3**

1: for \( i = 1, \ldots, n \) do
2: for \( j = i+1, \ldots, n \) do
3: Set \( \text{Max} := 0 \)
4: for \( t = 1, \ldots, i-1 \) do
5: if \( \min(a_t, a_i) < a_j < \min(a_t, a_j) \) and \( D[t, i] > \text{Max} \) then
6: Set \( \text{Max} := D[t, i] \)
7: \( D[i, j] = 1 + \text{Max} \)

4. (15 points) We are given a set of \( n \) jobs. The \( i \)-th job has a deadline \( d_i \), requires processing time of \( t_i \), and creates a profit of \( p_i \) only if it is finished before the deadline. The numbers \( d_i, t_i, \) and \( p_i \) are all positive integers. We have a single processor (that can start processing at time 0), and we want to select a subset of jobs that will create the maximum possible profit. Design and analyze an algorithm that performs this task.

**Solution:** Let \( T = \max d_i \) and \( K \) be the total number of jobs. Assume jobs are sorted by increasing time of deadline. For \( j = 0, \ldots, T \) and \( k = 1, \ldots, K \) let \( O[j, k] \) be the maximum profit that we can obtain until time \( j \) (that is the processor is not allowed to work after time \( j \)) considering only jobs which index is less or equal to \( k \). Note that our goal is to compute \( O[T, K] \) as there is no point in running the processor after time \( T \). As base cases we have
and our recurrence is defined as

\[ O[j, k] = \max \left\{ O[j - d_k, k - 1] + p_k, O[j, k - 1] \right\} \quad \text{if } d_k < j \leq d_k \]

Here \( O[j, k - 1] \) corresponds to the scenario where the job \( k \) is not considered and the second case is when the processor is processing job \( k \) and finishing it at time \( j \) (before its deadline \( d_k \)).

### Algorithm 3 Solution to problem 4

1: for \( k = 1, \ldots, K \) do
2: \hspace{1em} for \( j = 1, \ldots, T \) do
3: \hspace{2em} if \( d_k < j \) then
4: \hspace{3em} \( O[j, k] = O[j, k - 1] \)
5: \hspace{2em} else
6: \hspace{3em} \( O[j, k] = \max(O[j - d_k, k - 1] + p_k, O[j, k - 1]) \)

By keeping track of which element of the max operation was used to define the value of each \( O[j, k] \), we will be able to recover the subset of tasks to execute after our algorithm is complete.

Sorting our jobs by deadline takes \( O(K \log K) \). Our dynamic programming approach takes \( O(KT) \).

5. (20 points) We are given a directed graph with \( n \) nodes, and each node is labeled with the currency of a country (e.g. Canadian dollar, Yuan, Kyat, Euro, Ariary, etc). Every (directed) edge \( uv \) is labeled with a real number \( 0 < \alpha_{uv} \) signifying the fact that 1 unit of the currency \( u \) can be exchanged with \( \alpha_{uv} \) units of the currency \( v \). We want to see if we can make money by just buying and selling currencies. In other words, we want to find a sequence of currencies \( u_1, \ldots, u_k \) such that, we can start from one unit of \( u_1 \), and exchange it with \( \alpha_{u_1u_2} \) units of \( u_2 \), and exchange that with \( \alpha_{u_2u_3}\alpha_{u_1u_2} \) units of \( u_3 \), etc, and eventually exchange the amount of \( u_k \) that we obtained with \( u_1 \), so that we will end up with more than one unit of \( u_1 \).

Design and analyze a polynomial-time algorithm that tells us whether this is possible.

**Solution:** Note that we want to find a sequence \( u_1, \ldots, u_k \) such that \( \alpha_{u_1u_2} \ldots \alpha_{u_{k-1}u_k} \alpha_{u_ku_1} > 1 \). Taking the logarithm, this is equivalent to having

\[ -\log \alpha_{u_1u_2} - \log \alpha_{u_2u_3} - \ldots - \log \alpha_{u_ku_1} < 0. \]

Hence we can assign to every edge \( i_j \) a label of \( -\log \alpha_{u_iu_j} \) and use the Bellman-Ford algorithm to see if this weighted graph has a negative cycle.

6. (20 points) Design and analyze an efficient algorithm that given an undirected graph \( G = (V, E) \), and two nodes \( s, t \) finds the number of shortest paths from \( s \) to \( t \).

**Solution:** For every node \( u \) let \( N[u] \) be the number of shortest paths from \( s \) to \( u \). Note that \( N[s] = 1 \), and for every other vertex \( u \), in order to get to \( u \) we have to first get to a vertex \( v \) that has distance one less than the distance between \( s \) and \( u \). Hence \( N[u] = \sum_v N[v] \) where the sum is over all neighbours \( v \) of \( u \) that satisfy \( \text{dist}(s, v) = \text{dist}(s, v) - 1 \).

We run BFS to determine the distances of every vertex from \( s \), and then we compute \( N[u] \) iteratively for vertex with increasing distance from \( s \). Finally, we return \( N[t] \).