Worth: 5%

1. [20 marks]
We are given a list of $n > 0$ objects, say $X = (x_1, x_2, ..., x_n)$. The only operation we can do on these objects is equality testing (therefore we can not sort it). We must find all objects which occur in X strictly more than $n/3$ times. (We define $|X|$ to be this n.)

   (a) [3 points]
   Define M to be the maximum number of different objects (i.e., x_i and x_j with $x_i \neq x_j$) that can each appear in X strictly more often than $|X|/3$ times. (Note the minimum number of such frequent objects is zero.) Give the value of M and prove (carefully) that it is correct.

   (b) [12 points]
   Design a divide-and-conquer algorithm to solve the above problem. Present your algorithm as a pseudocode, and prove its correctness.

   (c) [5 points]
   Compute the complexity of your algorithm.

Solution.

   (a) Since M objects each appear strictly more than $|X|/3$ times, they together appear strictly more that $M|X|/3$ times. Since the list is of length $|X|$, we must have

$$|X| > M\frac{|X|}{3},$$

   i.e. $M < 3$. That is, the maximum number is $M = 2$.

   (b) In a Divide-and-Conquer approach, we typically divide the array into two halves L and R, solve the same problem for L and R using recursion, and combine the results to obtain a solution for the original array X. Observe that if we partition X into two halves L and R and an element a occurs more than $|X|/3$ times in X, then it must occur more than $|L|/3$ times in L or more than $|R|/3$ times in R or both. Otherwise, the number of occurrences of a in X is

$$n(a) \leq \frac{|L|}{3} + \frac{|R|}{3} = \frac{|X|}{3}.$$

   Hence, all we need to do is look for elements in L that occur more than $|L|/3$ times, and elements in R that occur more than $|R|/3$ times, which are our possible candidates and at most 4 in number, and then in the combine step count the number of occurrences of these candidates in the entire array. Also, observe that if the array X is of length 1 or of length 2, then each element of X satisfies the criteria of occurring strictly more than $|X|/3$ times. The full algorithm is presented on the next page.

   (c) Let $T(n)$ denote the complexity of this algorithm. Everything from Step 1 through 7 is $\Theta(1)$. Steps 8 and 9 are of complexity $T(n/2)$. The loops starting on lines 11 and 18 are $\Theta(1)$ since there are at most 2 elements in each of C_L and C_R. The loops starting on lines 13 and 20 are of complexity $\Theta(n)$. The remaining lines are all $\Theta(1)$. Thus, in total, we have

$$T(n) = 2T\left(\frac{n}{2}\right) + \Theta(n).$$

   By Case 2 of the Master Theorem, we obtain $T(n) = \Theta(n \lg n)$.

```
1: procedure MoreThanThirdWrapper(X)
2:     return MoreThanThird(X, 1, n)
```

```
1: procedure MoreThanThird(X, a, b)
2:     if b < a then
3:         return ∅
4:     else if b == a or b == a + 1 then
5:         return {X[a], X[a + 1]}
6:     else
7:         m := ⌊(a+b)/2⌋
8:         C_L = MoreThanThird(X, a, m)
9:         C_R = MoreThanThird(X, m + 1, b)
10:        C_X = {}
11:        for (i := 0; i < |C_L|; i++) do
12:            c := 0
13:            for (x ∈ X) do
14:                if x == C_L[i] then
15:                    c = c + 1
16:            if c > |X|/3 then
17:                C_X = C_X ∪ {C_L[i]}
18:        for (i := 0; i < |C_R|; i++) do
19:            c := 0
20:            for (x ∈ X) do
21:                if x == C_R[i] then
22:                    c = c + 1
23:            if c > |X|/3 then
24:                C_X = C_X ∪ {C_R[i]}
25:        return C_X
```

2. [20 marks]
Consider the problem of choosing cell phone tower locations on a long straight road. We are given a list of distances along the road at which there are customers' houses, say $\{x_k\}_{k=1}^{K}$. No towers have yet been built (so no customers currently have service). However, a survey of the road has provided a set of J possible tower locations, $\{t_j\}_{j=1}^{J}$, where these potential tower locations are also measured in terms of the distance along the road. (These indexed lists of house and tower locations might be in any order. You can assume that all these distances are integers.)

Each customer will get service (at home) if and only if they are within a range $R > 0$ of at least one cell phone tower that gets built, that is, the house at x_k will have service iff there exists a tower that has been built at some t_j with $|x_k - t_j| \leq R$.

You can assume that if all the towers were built then every home would get service. However, such a solution could be overly expensive for the phone company. How can we minimize the number of towers that need to be built but still provide service at every house? Note that a suitable solution may still leave some parts of the road without cell service, even though each customers' house will have service.

(a) [7 points]
Write a greedy algorithm for choosing the minimum number M of cell phone towers required, along with a suitable set of locations, say $T = \{t_{j(m)}\}_{m=1}^{M}$, such that every house has service. Present your algorithm as a pseudocode.

(b) [10 points]
Prove the correctness of your algorithm.

(c) [3 points]
Compute the complexity of your algorithm.

Solution.

(a) Here is our proposed greedy algorithm.

```
1:  procedure LocationOfTowers(X)
2:      sort the arrays {x_k | k = 1...K} and {t_j | j = 1...J} and assume they are in increasing order.
3:      T := {}
4:      LastAddedTowerIndex := −1
5:      for k = 0; k ≤ K; k++ do
6:          if T is empty or |x_k − t_LastAddedTowerIndex| > R then
7:              find tower t_j such that j > LastAddedTowerIndex, x_k ∈ [t_j − R, t_j + R], and t_j is as large as
    possible.
8:              if such a t_j exists then
9:                  T = T ∪ {t_j}
10:                 LastAddedTowerIndex = j
11:             else
12:                 print "No Solution Possible"
13:     return T
```

(b) We will prove by induction that for every $0 \leq k \leq K$, the set of towers $\mathcal{T}_k$ chosen by the greedy algorithm to cover all the houses at locations $\{x_1, \ldots, x_k\}$ is the same as the set of towers chosen by some optimal algorithm $\mathcal{O}$. And hence, the greedy algorithm is itself optimal.

Base Case: $k = 0$: This implies there are no houses to cover. So, the algorithm chooses $\mathcal{T}_k := \emptyset$, which is clearly a subset of every optimal solution.

Ind. Hyp.: For some $k > 0$, assume that there is an optimal solution $\mathcal{O}$ such that $\mathcal{T}_k$ and $\mathcal{O}$ choose the same towers to cover the houses at locations $\{x_1, \ldots, x_k\}$.

Ind. Step: At stage $k+1$, there are two possibilities: either,

Case 1: The towers in the set $\mathcal{T}_k$ already cover the house at location x_{k+1}; or,

Case 2: The towers in the set $\mathcal{T}_k$ do not cover the house at location x_{k+1}.

If Case 1 happens, then $\mathcal{T}_{k+1} = \mathcal{T}_k$, and hence, $\mathcal{O}$ and $\mathcal{T}_{k+1}$ agree on all the towers covering the houses at locations $\{x_1, \ldots, x_{k+1}\}$.

If Case 2 happens, then the greedy algorithm choses a tower t' such that $x_{k+1} \in [t' - R, t' + R]$ and t' is as large as possible. Then there are two further sub-possibilities: either,

SubCase 1: $\mathcal{O}$ choses the same tower t', in which case $\mathcal{O}$ and $\mathcal{T}_{k+1}$ agree on all chosen towers covering the houses at locations $\{x_1, x_2, \ldots, x_{k+1}\}$ (note that if $\mathcal{O}$ places any tower that is not in $\mathcal{T}_{k+1}$ but is to the left of t', then $\mathcal{O}$ is not an optimum solution since such a tower can be removed); or,

SubCase 2: $\mathcal{O}$ does not place a tower at location t'. Let y be the position of the leftmost tower in $\mathcal{O}$ that is not in $\mathcal{T}_{k+1}$; this tower must cover the house at location x_{k+1}, and hence must be to the left of t'. Define

$$\mathcal{O}' := \mathcal{O} \setminus \{y\} \cup \{t'\}.$$

Since a tower at position t' covers the house at location x_{k+1} and also every house to the right of the house at x_{k+1} that is covered by the tower at location y (and possibly more), it follows that $\mathcal{O}'$ is also a solution.
Since $\mathcal{O}$ and $\mathcal{O}'$ have the same size, it follows that $\mathcal{O}'$ is also optimal.
Moreover, $\mathcal{T}_{k+1}$ and $\mathcal{O}'$ share the same chosen towers that cover the houses at the locations $\{x_1, \ldots, x_{k+1}\}$, as desired.

(c) The complexity of the above algorithm is $\mathcal{O}(K \lg K) + \mathcal{O}(J \lg J) + \mathcal{O}(K + J)$. The first two terms are because of sorting the two arrays. Note that every line inside the loop on line 5 is $\mathcal{O}(1)$ except possibly line 7. Additionally, line 7 runs through the array $\{t_j \mid j = 1 \ldots J\}$ exactly once across all runs of the loop on line 5. Hence, the complexity.

3. [20 marks]
A waiter has n tables waiting to be served. The service time required by each table is known in advance: it is $t(i)$ minutes for table i. The waiter must serve each table one at a time, and once he starts serving a table he must completely finish serving it before starting to serve the next one.

The waiter wishes to serve the customers in an order that minimizes the total waiting time:

$$T = \sum_{i=1}^{n} (\text{time spent waiting by table } i)$$

His idea is to serve the tables in increasing order of $t(i)$. We can assume that the tables are pre-sorted in this order.

(a) [5 points]
Explain why the total waiting time can be written as $T = \sum_{i=1}^{n} (n - i)t(i)$.

(b) [10 points]
Give a proof that the waiter's algorithm is always optimal. You may assume the above formula is true, even if you weren't able to explain it.

(c) [5 points]
Compute the complexity of the waiter's algorithm.

Solution.

(a) When customer i is being served for a duration of $t(i)$, the remaining $n-i$ customers are waiting for the same duration of time. Thus, the total waiting time of all the customers is

$$T = \sum_{i=1}^{n} (n-i)t(i).$$

(b) Suppose the waiter's algorithm is not optimal. Then there exists an optimal schedule where some customer i is served before customer j ($i < j$) and $t(j) < t(i)$. Exchanging the two customers with each other, the total waiting time changes by

$$T_{new} - T_{old} = (n-j)t(i) + (n-i)t(j) - (n-i)t(i) - (n-j)t(j) = -jt(i) - it(j) + it(i) + jt(j) = (t(i) - t(j))(i-j) < 0.$$

The total waiting time of the new schedule is lower, a contradiction (since we assumed T_{old} is optimal).

(c) Since the tables are pre-sorted, the waiter simply goes through the tables in that order and serve them. Thus, the complexity of his algorithm is $\Theta(n)$.

4. [20 marks]
Suppose we are given a list of n integers, say $X = (x_1, \ldots, x_n)$. The problem is to find a longest subsequence of X, say $S = (x_{j_1}, x_{j_2}, \ldots, x_{j_K})$, where $|x_{j_{k+1}} - x_{j_k}| = 3$ for each $k = 1, 2, \ldots, K-1$, and also the length of such a sequence. Note that for S to be a ***subsequence*** of X, we must have $1 \le j_1 < j_2 < \cdots < j_K \le n$, i.e., the elements of S must be chosen in the same order as they appear in X.

For example, if $X = (1, 5, 3, 4, 2, 5, -1, 2)$, then $S = (1, 4)$ and $S = (5, 2, 5)$ are possible subsequences, but $S = (5, 2, -1, 2)$ is the longest subsequence with the desired property.

(a) [15 points]
Give a detailed dynamic programming algorithm to solve this problem. Follow the steps outlined in class, and include a brief (but convincing) argument that your algorithm is correct.

(b) [5 points]
What is the worst-case running time of your algorithm? Justify briefly. (For full credit, your algorithm must use at most $O(n)$ space and run in $O(n^2)$ time.)

Solution.

(a) Here is a DP solution in 5 steps:

Step I. Optimal substructure: For $1 \le k \le n$, define $\Theta(k)$:= length of longest such subsequence ending with x_k. Set $\Theta(0) := 0$.
Observe that the longest sequence ending with x_k must be one longer than the longest sequence ending at some suitable x_j ($j < k$), where "suitable" means that $|x_k - x_j| = 3$. Thus, we obtain the recurrence

$$\Theta(k) = \max\{\Theta(j) + 1 \mid 1 \le j < k \text{ and } |x_k - x_j| == 3\}.$$

Step II. Let M be an array for memoization, where $M[k]$ stores the length of a longest such subsequence ending with x_k.

Step III. Writing the recursive relation in terms of the array, we obtain

$$M[k] = \max\{M[j]+1 \mid 1 \le j < k \text{ and } |x_k - x_j| = 3\},$$

with $M[0] = 0$.

Step IV. We can now write an iterative bottom-up algorithm to compute M. Observe that for any i, $X[i-1] = x_i$.

```
1:  procedure LongestSubsequenceSizeDiffThree(X)
2:      n := |X|
3:      M := [0, 1, ..., 1] (n + 1 elements)
4:      MaxSize := 0
5:      for (k := 1; k ≤ n; k++) do
6:          for (j := 0; j < k; j++) do
7:              if M[j] + 1 > M[k] and |X[k − 1] − X[j − 1]| == 3 then
8:                  M[k] = M[j] + 1
9:          if M[k] > MaxSize then
10:             MaxSize = M[k]
11:     return MaxSize
```

Step V. Finally, we obtain a longest subsequence as follows:

```
1:  procedure LongestSubsequenceDiffThree(X, M)
2:      MaxSize := 0, MaxIndex := 0
3:      n := |X|
4:      for (k := 1; k < n; k++) do
5:          if M[k] > MaxSize then
6:              MaxSize = M[k]
7:              MaxIndex = k
8:      S := (X[MaxIndex − 1])
9:      L := MaxSize − 1 (L is the number of elements yet to be found)
10:     k := MaxIndex
11:     while L > 0 do
12:         for (j := 1; j < k; j++) do
13:             if M[j] == L and |X[k − 1] − X[j − 1]| == 3 then
14:                 S = S.prepend(X[j − 1])
15:                 L = L − 1
16:                 k = j
17:                 break
18:     return S
```

(b) It is easy to see that both algorithms (in Step 4 and Step 5) require $\mathcal{O}(n^2)$ time and $\mathcal{O}(n)$ space.

5. [20 marks]
Suppose we are given a $n \times n$ checkerboard with an integer-valued score written on each square, say $w(i, j)$, for $1 \le i, j \le n$. We are also given n pebbles. We obtain the score of a square only when we

place a pebble on it, and we wish to maximize the sum of these scores. The pebbles must be placed according to the following two rules:

(a) Any column can contain either zero or one pebble, not more, while rows can contain any number of pebbles (from zero up to n).

(b) If any two pebbles are placed on neighbouring columns, say columns j and $j+1$, with one pebble at (i, j), then the only possible locations for the other pebble is at either $(i-1, j+1)$ or $(i+1, j+1)$ (assuming, of course, that these later squares are still within the board).

A pebble placement that satisfies these rules is said to be feasible.

7	10	-5	4
6	2	3	10
5	-1	-5	-1
-1	7	-1	-7

For example, given the board in the figure, an optimal feasible placement is to put pebbles on rows and columns $(2,1)$, $(1,2)$ and $(2,4)$, for a score of $6+10+10=26$. Moreover, 26 is the maximum possible score for this board.

(a) [15 points]
Give a detailed dynamic programming algorithm to solve this problem. Follow the steps outlined in class, and include a brief (but convincing) argument that your algorithm is correct.

(b) [5 points]
What is the worst-case running time of your algorithm? Justify briefly. (For full credit, your algorithm must use at most $O(n^2)$ space and run in $O(n^3)$ time.)

Solution.

(a) Here is a DP solution in 5 steps:

Step I. Optimal substructure: For $(i, j) \in \{0, \ldots, n\} \times \{0, \ldots, n\}$, define

$$\Theta(i,j) := \begin{cases} \text{maximum score of feasibly placing pebbles only on the first } j \text{ columns with a pebble at location } (i,j) & \text{if } i > 0, j \geq 1 \\ \text{maximum score of feasibly placing pebbles only on the first } j\text{-1 columns} & \text{if } i = 0, j \geq 1 \\ 0 & \text{if } i = 0, j = 0 \\ -\infty & \text{if } i > 0, j = 0 \end{cases}$$

The last entry is for illegal entries. Set $w(0, j) := 0$, score for no pebble in column j.

Now observe that if we place a pebble at location (i, j), then there are only three choices for column $j-1$: we can either put a pebble at location $(i-1, j-1)$ (provided $(i-1, j-1)$ is on the board), or put a pebble at location $(i+1, j-1)$ (provided $(i+1, j-1)$ is on the board), or don't place any pebble in column $j-1$ at all. Our optimal solution will be the best of these three options.
And if we don't place any pebble in column j, then we can place a pebble at any row of column $j-1$ depending on where we get the maximum score.

Combining all these observations, we obtain the recurrence as follows:

$$\Theta(i,j) = \max_{k \in N(i)} \{\Theta(k, j-1) + w(i,j)\},$$

where

$$N(i) = \begin{cases} \{0, i-1, i+1\} \cap \{0, 1, \ldots, n\} & \text{if } i > 0 \\ \{0, 1, \ldots, n\} & \text{if } i = 0 \end{cases}$$

Step II. Let M be an array for memoization, with $M[i,j] = \Theta(i,j)$ for $0 \le i \le n, 0 \le j \le n$.

Step III. Writing the recurrence in terms of the array M, we get

$$M[i,j] = \max_{k \in N(i)} \{M[k, j-1] + w(i,j)\},$$

where

$$N(i) = \begin{cases} \{0, i-1, i+1\} \cap \{0, 1, \ldots, n\} & \text{if } i > 0 \\ \{0, 1, \ldots, n\} & \text{if } i = 0 \end{cases}$$

Step IV. We can now write an iterative bottom-up algorithm to compute M.

```
1:  procedure CheckerBoardMaxScore(n, w)
2:      Define array M of size (n+1) × (n+1)
3:      Set M[0,0] := 0 and M[i,0] := −∞ for i > 0
4:      for (j := 0; j ≤ n; j++) do
5:          for (i := 0; i ≤ n; i++) do
6:              if i == 0 then
7:                  N = {0, 1, ..., n}
8:              else
9:                  N = {0, i−1, i+1} ∩ {0, 1, ..., n}
10:             Max := −∞
11:             for k ∈ N do
12:                 if M[k, j−1] + w[i, j] > Max then
13:                     Max = M[k, j−1] + w[i, j]
14:             M[i, j] = Max
15:     MaxScore := −∞
16:     for (k = 0; k ≤ n; k++) do
17:         if M[k, n] > MaxScore then
18:             MaxScore = M[k, n]
19:     return MaxScore
```

Step V. Finally, we write an algorithm to find out which locations to put the pebbles on. Since we are only allowed to put at most one pebble in each column, we will output a list L of size n, where $L[j]$ stands for the row number in column j where we should put a pebble. Note, $L[j] = 0$ means no pebble in column j.

```
1:  procedure CheckerBoardPebbleLocations(M, MaxScore)
2:      Define an empty list L
3:      S := MaxScore
4:      for (j = n; j ≥ 1; j- -) do
5:          if j == n then
6:              i = [ find k ∈ {0,...,n} such that M[k,n] == S ]
7:          else
8:              i = [ find k ∈ N(i) such that M[k,j] == S ]
9:          L.prepend(i)
10:         S = S − w(i,j)
11:     return L
```

(b) Clearly the time complexities of Steps IV and V are $\Theta(n^3)$ and $\Theta(n^2)$, respectively. And the space complexity for both parts is $\Theta(n^2)$.