Problem 1

Notation $a \leftrightarrow b$ means that a is matched to b. $a <_b c$ means that b likes c more than a. Equality indicates a tie.

Strong instability

Yes, there does always exist a perfect matching without a strong instability.

Intuition We can resolve ties by ordering the persons involved arbitrarily. I.e. when m_1 and m_2 are tied for w_1, we can just assume that w_1 likes m_1 better than m_2. Since m_1 and m_2 were tied before, w_1 will never complain about our choice. The Gale-Shapley algorithm could then be run without modifications and is guaranteed to return a perfect stable matching.

Proof We show that the Gale-Shapley algorithm always generates a perfect matching without a strong instability, even if there may be ties. We will prove this by contradiction.

We make a small modification to the algorithm: if there are multiple choices for a woman to propose to a man (because there are multiple men that she likes best), she randomly proposes to one.

Also, note that men do not switch for a women that are tied with the current partner.

- We assume, that the output of the algorithm contains a strong instability. I.e. $m_1 \leftrightarrow w_1$ and $w_2 \leftrightarrow w_2$, but $m_1 <_{w_1} m_2$ and $w_2 <_{m_2} w_1$.
- Therefore, w_1 must have proposed to m_2 before she proposed to m_1[1].
- $m_1 \leftrightarrow w_1$, so one of the following must be true.
  - m_2 already had a better partner w_i. Eventually, m_2 switched for w_1. Therefore, $w_2 \geq_{m_2} w_i >_{m_2} \ldots >_{m_2} w_1$. This violates our assumption that $w_2 <_{m_2} w_1$.
  - m_2 accepts w_1, but eventually switches to w_2. Therefore, there must be women, including w_2, that m_2 likes better. Therefore, $w_2 \geq_{m_2} w_i >_{m_2} \ldots >_{m_2} w_1$. This violates our assumption that $w_2 <_{m_2} w_1$.

[1] m_1 and m_2 are not tied (see assumption).

  - Note that w_1 and w_2 cannot be tied, since this would not impose a strong instability.

- Our assumption is wrong. There can be no strong instability.

Weak instability

No, there is not always a perfect matching without a weak instability. We show this by providing an example in which all perfect matchings have a weak instability.

$$w_1 =_{m_1} w_2$$
$$w_1 =_{m_2} w_2$$
$$m_1 >_{w_1} m_2$$
$$m_1 >_{w_2} m_2$$

We examine the two possible matchings.

- $m_1 \leftrightarrow w_1$ and $m_2 \leftrightarrow w_2$ contains a weak instability: w_2 likes m_1 more than her current partner m_2 and m_1 does not care (both women are tied).

- $m_1 \leftrightarrow w_2$ and $m_2 \leftrightarrow w_1$ contains a weak instability: w_1 likes m_1 more than her current partner m_2 and m_1 does not care (both women are tied).

Therefore, there does not always exist a perfect matching without a weak instability.

Problem 2

For a woman, it is not possible to end up with a true better man by lying about her preferences.

- Let w be a woman with the true preference $m_1 >_w m_2$. Let us assume that the Gale-Shapley algorithm creates $m_{truth} \leftrightarrow w$ and that w pretends that $m_2 >_w^{lie} m_1$. We compare the results of the algorithm with and without the lie.

- **Case 1:** $m_1 < m_{truth}$ **and** $m_2 < m_{truth}$
  Original proposing sequence: $(\ldots, m_{truth}, \ldots, m_1, \ldots, m_2, \ldots)$
  New proposing sequence: $(\ldots, m_{truth}, \ldots, m_2, \ldots, m_1, \ldots)$
  Both m_1 and m_2 are actually and *after lying* lower on w's preference list than her matched partner m_{truth}. Therefore, w is matched to m_{truth} before and after the lie, because, in both cases, w does not even propose to m_1 or m_2, since she was matched with a better partner before and he did not leave w[2].

- **Case 2:** $m_1 > m_{truth}$ **and** $m_2 > m_{truth}$
  Original proposing sequence: $(\ldots, m_1, \ldots, m_2, \ldots, m_{truth}, \ldots)$
  New proposing sequence: $(\ldots, m_2, \ldots, m_1, \ldots, m_{truth}, \ldots)$
  Both m_1 and m_2 are actually and *after lying* higher on w's preference list than her matched partner m_{truth}. Therefore, w is matched to m_{truth} before and after the lie, because, in both cases, w proposes to m_1 and m_2 and both will either reject her or first accept her and leave her later.

  - At some point, w proposes to m_2. We know that $w \leftrightarrow m_{truth}$ in the first run of the algorithm, although w proposed to m_2 before. Therefore, m_2 either already had a better partner or he accepted w and then switched to another woman. This will also happen after lying, since w's lie does not affect the proposing sequence of the other women.
  - The same argument holds true for m_1.

- **Case 3:** $m_1 > m_{truth}$ **and** $m_2 < m_{truth}$
  Original proposing sequence: $(\ldots, m_1, \ldots, m_{truth}, \ldots, m_2, \ldots)$
  New proposing sequence: $(\ldots, m_2, \ldots, m_{truth}, \ldots, m_1, \ldots)$
  When w lies, the following is true.

  - w will not be matched with a man she proposed to before m_2, because these men eventually rejected her in the first run of the algorithm.
  - When w proposes to m_2, he might accept her and stay with her. In that case, w ends up with a worse partner than without lying.
  - In case m_2 rejects w or leaves her later, w will propose to all men until m_{truth} will eventually accept her and stay with her. All the

[2] In general, the ordering of men on w's preference list below m_{truth} does not matter.

men in between will reject her or leave her, because that is what they did in the first run of the algorithm.

- **Case 4:** $m_1 < m_{truth}$ **and** $m_2 > m_{truth}$
  Not possible, since $m_1 < m_{truth} < m_2$ violates our assumption that $m_1 > m_2$.

- **Case 5:** $m_2 = m_{truth}$ **and** $m_1 > m_{truth}$
  Original proposing sequence: $(\ldots, m_1, \ldots, m_{truth}, \ldots)$
  New proposing sequence: $(\ldots, m_{truth}, \ldots, m_1, \ldots)$
  w changes the ordering in such a way that she proposes to her matched partner earlier and to a better partner later. m_{truth}'s predecessors will (eventually) reject w, but m_{truth} will accept w and stay with her, since he did not get a better proposal in the first run.

- **Case 6:** $m_2 = m_{truth}$ **and** $m_1 < m_{truth}$
  Not possible, since $m_1 < m_2$ violates our assumption that $m_1 > m_2$.

- **Case 7:** $m_1 = m_{truth}$ **and** $m_2 < m_{truth}$
  Original proposing sequence: $(\ldots, m_{truth}, \ldots, m_2, \ldots)$
  New proposing sequence: $(\ldots, m_2, \ldots, m_{truth}, \ldots)$
  All men before m_2 will reject w's proposal or leave her later, since that is what they did in the first run. w proposes to m_2 before she proposes to m_{truth}. m_2 might accept her proposal and stay with her. In that case, w ended up with a worse partner. In case m_2 rejects her (eventually), w will propose to all men between m_2 and m_{truth}. These men will reject her, because they did the same in the first run. m_{truth} will accept w and stay with her.

- **Case 8:** $m_1 = m_{truth}$ **and** $m_2 = m_{truth}$
  w did not lie.

In every case, w did either end up with the same partner again or get a worse partner. Therefore, w cannot improve her matching by lying.

Problem 3

Summary

The key idea is to calculate all values F_j together instead of separately. When we take a look at the formula for F_j, we see that C and C_j are constants that are multiplied with the two sums. The difficult part is to calculate $\sum_{i<j} \frac{q_i}{(j-i)^2} - \sum_{i>j} \frac{q_i}{(j-i)^2}$ efficiently. We can generate two polynoms $Q(x)$ (for the q_i) and $M(x)$ (for the denomiators) of less than degree $2n$ such that their multiplication contains the values F_j. This can be done efficiently with the algorithm discussed in the lecture. Then we multiply every coefficient of the resulting polynom with the constants C and q_j.

Basic Idea

- Calculate all F_j together instead of separately.
- Generate polynoms
  - $Q(x) = \sum_{i=1}^{n} q_i x^{n-i}$
  - $M(x) = \sum_{i=1}^{n-1} -\frac{x^{n-1+i}}{i^2} + \frac{x^{n-1-i}}{i^2}$
- Calculate $F(x) = Q(x) \cdot M(x)$ using the algorithm discussed in the lecture. Degree of both polygons is less than $2n$. Runtime $\mathcal{O}(2n \log 2n)$.
- Extract coefficients $F_j = f_j$ from $F(x)$ at positions $j = 2n - 1 - i$ for $i = 1 \dots n$.

Full Algorithm

We can rewrite F_j as follows.

$$F_j = C \cdot q_j \cdot \left(\sum_{i<j} \frac{q_i}{(j-i)^2} - \sum_{i>j} \frac{q_i}{(j-i)^2} \right)$$

Now, we geneate the polynoms $Q(x)$ and $M(x)$ as presented in the previous section. For instance, for $n = 4$, we generate the following polynoms.

$$Q(x) = q_1 x^3 + q_2 x^2 + q_3 x + q_4$$

$$M(x) = -\frac{1}{9}x^6 - \frac{1}{4}x^5 - x^4 + x^2 + \frac{1}{4}x + \frac{1}{9}$$

We use the method discussed in the lecture to multiply these two polynoms. For polynoms of degree m this takes $\mathcal{O}(m \log m)$ time. $\deg(Q(x)) = n - 1$ and $\deg(M(x)) = 2 \cdot (n-1) < 2n$. Therefore, the multiplication takes $\mathcal{O}(2n \log 2n)$ time.

Note: the algorithm for multiplying two polynomials was designed for polynoms whose degree is a power of 2. Therefore, we might have to extend the polynoms, resulting in a polynom of degree $2n$. Therefore, the multiplication takes $\mathcal{O}(4n \log 4n) = \mathcal{O}(n \log n)$ time.

For the multiplication, we assume that $Q(x)$ has the same degree as $M(x)$ by adding the missing powers of x with a coefficient of zero (e.g. $0x^6$ in the example). We also add missing powers of x inside $M(x)$ (e.g. $0x^3$ in the example).

$$M(x) \cdot Q(x) = \ldots + x^6(-q_2 - \frac{q_3}{4} - \frac{q_4}{9}) + x^5(q_1 - q_3 - \frac{q_4}{4}) +$$
$$x^4(\frac{q_1}{4} + q_2 - q_4) + x^3(\frac{q_1}{9} + \frac{q_2}{4} + q_3) + \ldots$$

The example above shows only the coefficients of x^k with $k = 3 \ldots 6$. In general, we are interested only in the coefficients of x^k with $k = 2n - 1 - i$ and $i = 1 \ldots n$. The other coefficients are calcuated by the algorithm, but we do not need them.

We can now generate the term $F_j = C \cdot q_j \cdot c_{2n-1-j}$, where c_i is the coefficient of x^i in $M(x) \cdot Q(x)$. This requires $\mathcal{O}(n)$ multiplications in total.

Pseudo Code

$n \leftarrow Q.size()$
$M \leftarrow \text{Array}[2n-1]$
$result \leftarrow \text{Array}[n]$
for $i = 1$ to n **do**
    $M[n-1+i] \leftarrow i^2$
    $M[n-1-i] \leftarrow -i^2$
end for
$d \leftarrow 2^{\lceil \log_2 \deg M \rceil}$
$Q \leftarrow Q$ with d coefficients (fill with zeros)

$M \leftarrow M$ with d coefficients (fill with zeros)
$p \leftarrow mult(M, Q)$
for $i = 1$ to n **do**
    $result[i-1] = p[2n-1-i]$
end for
    return *result*

Runtime complexity

- Generate polynom $Q(x)$: $\mathcal{O}(n)$
- Generate polynom $M(x)$: $\mathcal{O}(2n)$
- Convert polynoms $Q(x)$ and $M(x)$ to polynom with same number of coefficients, where the number of coefficients is a power of 2: $\mathcal{O}(4n)$ each (next power of 2 might be almost $2n$)
- Multiply polynoms (using FFT): $\mathcal{O}(4n \log 4n)$
- Extract coefficients from resulting polynom: $\mathcal{O}(n)$

Therefore, the runtime complexity is $\mathcal{O}(4n \log 4n) = \mathcal{O}(n \log n)$.

Problem 4

Basic Idea

- Given an arbitrary starting vertex, a local minimum is the end of a decreasing path, i.e. every next vertex is smaller than the current vertex.
- Find the smallest vertex v in the middle column.
- Check if this smallest vertex is a local minimum. If not, there must be a local minimum on that side of the column where v's (left/right) neighbor is smaller than v (could be both sides). A smallest path starting from v's neighbor will never cross this column.
- Find the smallest vertex w in the middle row of the side that we chose in the last step.

- Check if w is a local minimum. If not, there must be a local minimum inside the quarter sector containing v's chosen neighbor if $w > v$. Otherwise, there must be a local minimum in the quarter sector where w's (upper/lower) neighbor is smaller than w.
- We continue recursively on a smaller grid.

Full Algorithm

1. Find cell $\left(\lceil \frac{n}{2} \rceil, i\right)$, $1 \leq i \leq n$ that has a minimal value $x_{\lceil \frac{n}{2} \rceil, i}$.
2. Check $\left(\lceil \frac{n}{2} \rceil, i\right)$'s neighbors. If they are both greater than $x_{\lceil \frac{n}{2} \rceil, i}$, we return the local minimum.
3. Otherwise, there must be a local minimum on the side that contains the neighbor vertex P that has a smaller value than $x_{\lceil \frac{n}{2} \rceil, i}$. If both neighbors have a smaller value, then both sides contain a local minimum and we may choose P arbitrarily.
   Proof: Starting with a vertex of value x_i, we can find a local minimum by choosing a neighbor that has a smaller value $x_j < x_i$. We must eventually reach a local minimum, because all values x_k are distinct. Therefore, there must always be a smaller neighbor or, in case we reached the local minimum, all neighbors are greater. The path we consider right now starts at P with a value that is smaller than $x_{\lceil \frac{n}{2} \rceil, i}$. This x_P is smaller than the smallest number of the column. We can be sure that this (decreassing-value) path will never cross this column again. Otherwise, the path would hit a greater value, since all vertices on the column have a greater number than x_P.
4. Find cell $\left(j, \lceil \frac{n}{2} \rceil\right)$, with $1 \leq j \leq \lceil \frac{n}{2} \rceil - 1$ if we just chose the left side, or otherwise $\lceil \frac{n}{2} \rceil + 1 \leq j \leq n$ (if we chose the right side), such that $x_{j, \lceil \frac{n}{2} \rceil}$ is minimal.
5. Check $\left(j, \lceil \frac{n}{2} \rceil\right)$'s neighbors. If they are both greater than $x_{j, \lceil \frac{n}{2} \rceil}$, we return the local minimum.
6. We compare $x_{\lceil \frac{n}{2} \rceil, i}$ and $x_{j, \lceil \frac{n}{2} \rceil}$.
   (a) If $x_{\lceil \frac{n}{2} \rceil, i} < x_{j, \lceil \frac{n}{2} \rceil}$, then the path starting at P will not cross the row, because all vertices on the row have a greater value than the

x_P. Therefore, we know that there must be a local minimum in the (square) sector that contains P. We continue the algorithm in that sector recursively. The new sector's size is equal to or smaller than $\lceil \frac{n}{2} \rceil \times \lceil \frac{n}{2} \rceil$.

(b) If $x_{\lceil \frac{n}{2} \rceil, i} > x_{j, \lceil \frac{n}{2} \rceil}$, then we take one of $\left(j, \lceil \frac{n}{2} \rceil\right)$'s neighbors whose value is smaller than $x_{j, \lceil \frac{n}{2} \rceil}$ and call it Q[3]. We know that the path starting from Q will cross neither the half-row nor the column (same reason as in the proof above). Therefore, we know that there must be a local minimum in the (square) sector that contains Q. We continue the algorithm in that sector recursively. The new sector's size is equal to or smaller than $\lceil \frac{n}{2} \rceil \times \lceil \frac{n}{2} \rceil$.

- $n = 1$ is the base case for the recursion. In that case, the grid consists of only one cell that must be a local minimum.

Pseudo Code

This is the pseudo code for *findMin*(G, n), where G is the grid graph and n is its size in one dimension.

```
if n = 1 then                                              ▷ base case
    output(1, 1)        ▷ keep track of acutal indices, see comment below
end if
c_x ← ⌈n/2⌉
c_v ← ∞
for i = 1 to n do
    x ← G[c_x, i]
    if x < c_v then                                        ▷ step 1
        c_v ← x
        c_y ← i
    end if
end for
n_l ← G[c_x − 1, c_y]
n_r ← G[c_y + 1, c_y]
if n_l > x ∧ n_r > x then                                  ▷ step 2
    output(c_x, c_y)                                       ▷ keep track of actual indices
```

[3]There must always exist such a vertex. Otherwise we would already have returned the local minimum.

else
    $direction_x \leftarrow -1$ **if** $n_l < x$ **else** 1
end if
$r_y \leftarrow \lceil \frac{n}{2} \rceil$
$r_v \leftarrow \infty$
for $i = \lceil \frac{n}{2} \rceil + direction_x; i > 0 \wedge i < n + 1; i \leftarrow i + direction_x$ **do**
    $x \leftarrow G[i, r_y]$
    if $x < r_v$ **then** ▷ step 4
        $r_v \leftarrow x$
        $r_x \leftarrow i$
    end if
end for
$n_u \leftarrow G[r_x, r_y - 1]$
$n_d \leftarrow G[r_x, r_y + 1]$
$c_{end} \leftarrow 1$ **if** $n_l < x$ **else** n
if $n_u > x \wedge n_d > x$ **then** ▷ step5
    output(r_x, r_y) ▷ keep track of actual indices
else if $c_v < r_v$ **then** ▷ step 6a
    $r_{end} \leftarrow 1$ **if** $c_y < \lceil \frac{n}{2} \rceil$ **else** n
else ▷ step 6b
    $r_{end} \leftarrow 1$ **if** $n_d < x$ **else** n
end if
    return findMin$(G[\lceil \frac{n}{2} \rceil : c_{end}, \lceil \frac{n}{2} \rceil : r_{end}], \lceil \frac{n}{2} \rceil)$

In the pseudo code, $G[a : b, c : d]$ means that we copy the array rectangle horizontally from index a to b and vertically from c to d. The resulting array begins at index 1. In the actual implementation of the algorithm, we would not do this because it is too expensive. We can just provide the coordinates of the new array segment that we are working on. The only reason for choosing this way in the pseudo code is to keep it readable and understandable. In the runtime complexity analysis, I will not account for copying the array.

It is also important to remember, that the output of the indices of the pseduo code algorithm does not keep track of the actual indices. I.e. every recursive call of the algorithm gets an array whose first element is at position $(1, 1)$, although the real position of this first element might be something else (because it was copied in the example in order to keep the pesudo code free of index number crunching).

All out-of-bounds accesses will return ∞. I.e. $G[0, 2] = G[n + 1, 1] = \infty$.

Runtime complexity

- We assume that the grid G is a square grid with a height and length of n.
- Finding the minimum of G's middle column: n probes.
- Checking if the column's minimum is a local minimum: 2 probes.
- Finding the minimum of the middle column: $\lceil \frac{n}{2} \rceil - 1 = \mathcal{O}(n)$ probes[4].
- Checking if the row's minimum is a local minimum: 2 probes.
- We need $\mathcal{O}(n)$ probes per recursion/conquer step. Every divide step divides n by half[5]. Therefore this is the runtime complexity for the algorithm: $T(n) = T(\frac{n}{2}) + \mathcal{O}(n)$. The base case takes a constant amount of time ($T(1) = c$) We can show that $T(n) = \mathcal{O}(c \cdot n)$ by using the recursion tree method. Every level contains exactly one problem and the problem size is cut in half on every next level. Therefore, the problem complexity can be described by the decreasing geometric series: $T(n) = \sum_{i=1}^{\log n} \frac{c \cdot n}{2^i} \leq c_2 \cdot n$.[6]

Therefore, the runtime complexity of the algorithm is $\mathcal{O}(n)$.

Problem 5

Basic Idea

- We sort all lines by their slopes.
- We use divide-and-conquer and generate two equal-sized subproblems by splitting the lines in the middle. Note, that the sorting is preserved.
- The algorithm is supposed to return the envelope that contains the visible lines only.

[4]Remember that we already split the grid vertically at this point

[5]If the n is odd, we take $\lceil \frac{n}{2} \rceil$. This does not affect the runtime complexity asymptotically.

[6]We can assume that it is an infinite decreasing geometric series.

- In the conquer step, we take two envelopes and combine them. Therefore, we find the single intersection point of the envelopes and from there on, we take the line segments that are on top. Before, we take the line segments from the other envelope. The intersection point can be found in linear time with a scan line algorithm that begins at $-\infty$ and stops at the beginning of every new line segment in either one of the two envelopes.

Full Algorithm

- The algorithm gets an array of line equations as input, and the line equations must already be sorted by their slope value. If that is not the case, this can be done in a preprocessing step in $\mathcal{O}(n \log n)$ if n is the number of line equations.

- The algorithm returns a list of visible line equations and their intersection points. Together, these form line segments, where each line equation is only used between the two intersection points with the predecessor and the successor line equation. Note, that the left-most and the right-most lines have only one intersection and go to $-\infty$ and ∞ respectively.

- We divide the list of equations in the middle and work on both list recursively (divide and conquer).

- The conquer step works as follows.

  - We need to merge the two sequences of line segments (also called *envelopes*) by finding their intersection point and creating one single list of line segments. During this step, some of the lines might become invisible and are therefore discarded. This can be done in $\mathcal{O}(n)$.
  - Let A be the line segments from the first recursive divide call and B be the line segments from the second recursive divide call.
  - At some point, a segment from A might intersect with a segment from B. At that point, we combine A and B by saving the intersection point and taking all previous line segments from A and

taking all following line segments from B, if A was on top before and B is on top later[7].

- Note, that there can only be one single intersection of A and B, because A and B are sorted by their slopes and all slopes in A are lower than all slopes in B. If there were a second intersection point, the two envelopes had to intersect a second time. Therefore, A's or B's slopes would have to decrease again or A's slopes would have to become greater than a slope in B. This contradicts the fact, that A and B are sorted.
- We can merge both envelopes in linear time by using a scan line algorithm. We start at $x = -\infty$ and determine the value of the first line equation in both envelopes[8]. By doing this, we know which of the two envelopes is on top. Then we move to the next closest intersection point (on the x-axis) in the two envelopes[9]. Note, that the slope of an envelope can only change at these intersection points. We evaluate the other envelope at this intersection point[10]. If from now on, the other line segments/the other envelope are on top (e.g. before, A was on top, but now B is on top)[11], then A and B intersect and we combine them as described before. We return the combined envelopes.

Runtime Complexity

- Sorting by slopes: $\mathcal{O}(n \log n)$
- Scan line algorithm for combining two envelopes (conquer step): $\mathcal{O}(n)$
- Full recursion with divide and conquer: $T(n) = 2 \cdot T(\frac{n}{2}) + \mathcal{O}(n)$ and the base cases $T(1) = T(2) = \text{const}$. We know that $T(n) = \mathcal{O}(n \log n)$.

Therefore, the overall runtime complexity is $\mathcal{O}(n \log n)$.

[7] Otherwise, we take all previous line segments from B, add the intersection point, and take all following line segments from A.

[8] Note, that the first equation does not have an intersection point to its left.

[9] Both envelopes have different intersection points and we do not care where the intersection point comes from.

[10] We know the segments of the other envelope and where they begin and end.

[11] We have to evaluate both envelopes.