Solutions to Problem Set 1

1. **(Description on the Problem Set)**

(a) We use dynamic programming to solve this problem. Let \( f(n) \) be the number of ways we can tile a \( 3 \times n \) rectangle. This is the solution we want to compute. Moreover, let \( g(n) \) be the number of ways we can tile a \( 3 \times n \) rectangle if we remove one of its right corner squares. We can write the recursive relations:

\[
\begin{align*}
  f(n) &= f(n-2) + 2g(n-1) \\
  g(n) &= f(n-1) + g(n-2)
\end{align*}
\]

The boundary values are \( f(0) = 1, f(1) = 0, g(0) = 0, g(1) = 1 \). Note also that for every odd \( n \) there is no valid tiling.

**Complexity:** computing the \( f \) and \( g \) arrays of \( n \) values requires at most \( O(n) \) steps.

**Correctness:** Using recursion, the only way to tile the right border of the rectangle is using the configuration \((a), (b)\) and the flipped (on the horizontal axis) version of \((b)\).

\[
(a) = f(n-2) \quad (b) = g(n-1)
\]

A similar statement can be easily proved for obtaining \( g(n) \) exactly in terms of \( f(n-1) \) and \( g(n-2) \).

(b) We define the following recursive function \( F(n, \vec{x}) \), where the vector \( \vec{x} \in \{0, 1, 2\}^k \), as the number of ways to tile a grid that has \( n - 2 + \vec{x}_i \) cells in the \( i \)-th row (rows are assumed always left-justified). Let \( i^* \) be the first row starting from the top of the grid for which \( \vec{x}_{i^*} = 2 \). We can express the recursive relation as follows (note that the first case is when \( i^* \) does not exist):

\[
F(n, \vec{x}) =
\begin{cases}
  F(n-1, \vec{x} + \vec{1}), & \text{if } \forall i, \vec{x}_i \neq 0 \\
  F(n, \vec{x} \text{ with } \vec{x}_{i^*} = 0), & \text{if } \vec{x}_{i^*} = 2 \text{ and } \vec{x}_{i^*+1} \neq 2 \\
  F(n, \vec{x} \text{ with } \vec{x}_{i^*} = 0) + F(n, \vec{x} \text{ with } \vec{x}_{i^*} = \vec{x}_{i^*+1} = 1), & \text{if } \vec{x}_{i^*} = \vec{x}_{i^*+1} = 2
\end{cases}
\]

The first case is trivial to prove (note that incrementing \( \vec{x} \) by 1 at each entry still maintains the vector in \( \{0, 1, 2\}^k \)). The second case considers when the row \( i^* + 1 \) does not have 2 blocks after the first \( n - 2 \): in this case the only way to tile the last block in the \( i^* \)-th row is to place a \( 2 \times 1 \) tile horizontally. Similarly, the third case assumes that also the \((i^*+1)\)-th row has 2 ending blocks: thus the only 2 ways to tile the last block on the \( i^* \)-th row is to place a tile horizontally and vertically starting from that last block.

We compute this recursion by first increasing the value of \( n \) and then increasing \( \sum_i \vec{x}_i \). Since for each value of \( n \), we need to cover \( 3^k \) configuration of \( \vec{x} \), the complexity is \( \exp(k) \cdot n \). Hence, the algorithm complexity is exponential in \( k \).
(c) We rewrite the recursive relations using matrix multiplication as:

\[
F^n \equiv \begin{bmatrix} f(n) \\ f(n-1) \\ g(n) \\ g(n-1) \end{bmatrix} \cdot \begin{bmatrix} 0 & 1 & 2 & 0 \\ 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 \end{bmatrix} = \begin{bmatrix} f(n+1) \\ f(n) \\ g(n+1) \\ g(n) \end{bmatrix} \equiv F^{n+1}
\]

Thus \( F^k = F^1 \cdot A^{k-1} \) where \( F^1 = [0, 1, 1, 0]^T \) and \( A \) is the \( 4 \times 4 \) integer matrix we just found above. At this point we can already use the matrix exponentiation algorithm to compute \( A^{k-1} \) which requires \( O(\log k) \) steps. To obtain the \( O(1) \) complexity, we need to use the eigenvalues formula for the \( n \)-th power of the matrix \( A \). The closed formula for computing the solution is

\[
f(n) = \frac{(1 + (-1)^n) \times ((2 - \sqrt{3})^{n/2} \times (-1 + \sqrt{3}) + (1 + \sqrt{3}) \times (2 + \sqrt{3})^{n/2})}{4\sqrt{3}}
\]

2. (Description on the Problem Set)

The idea is that a sub-array of at least \( k \) elements must have an average smaller or equal to the maximum average sub-array. Thus we use a binary search to get close to our best \( k \) (or more) elements average.

\textbf{Algorithm 1 Max Average}

\begin{verbatim}
1: procedure MaxAve(A) \hspace{1em} \triangleright Input array
2:     Compute the max average \( M^* \) over all sub-arrays of \( A \)
3:     \text{min} = 0; \text{max} = \text{M}^*
4:     \textbf{while} \text{max} - \text{min} > 1/n^2 \textbf{do}
5:         \text{mid} = (\text{min} + \text{max})/2
6:         \textbf{if} there is a subarray of size at least \( k \) with average \( \geq \text{mid} \) \textbf{then}
7:             \text{min} = \text{mid}
8:         \textbf{else}
9:             \text{max} = \text{mid}
10: \textbf{end if}
11: \textbf{end while}
12: \textbf{end procedure} \hspace{1em} \triangleright \text{Solution is } \text{mid}
\end{verbatim}

Note that step 2 can be computed in \( O(n) \). We need to show that step 6 can also be computed in linear time. We create a new array \( B \) where \( B[i] = A[i] - \text{mid} \). Now we look for the maximum sum \( S \) sub-array of length at least \( k \). If \( S \geq 0 \) then we know that there is a sub-array \( A' \) of at least \( k \) elements with average greater than \( \text{mid} \). In fact, assume \( \text{w.l.o.g.} \) that the sub-array \( A' \) goes from \( A[p] \) to \( A[p+k-1] \), then
\[
\frac{1}{k} \sum_{i=p}^{p+k-1} A[i] \geq \text{mid}
\]

\[
p+k-1 \sum_{i=p}^i A[i] \geq k \cdot \text{mid}
\]

\[-k \cdot \text{mid} + \sum_{i=p}^{p+k-1} A[i] \geq 0
\]

\[
p+k-1 \sum_{i=p}^i (A[i] - \text{mid}) \geq 0
\]

\[
S \geq 0
\]

Thus, the problem reduces in finding a sub-array \(B'\) (of \(B\)) of length at least \(k\) such that the sum of its elements is at least 0. This problem can be computed in \(O(n)\) using a modified version of Kadane’s D.P. algorithm: we use the following recursive relation

\[
C[k] = \sum_{i=1}^{k} B[i], \text{ and for } j > k \text{ we have } C[j] = \max\{C[j-1] + B[j], C[j-1] + B[j] - B[j-k]\}
\]

**Complexity:** at each step of the binary search we perform only linear computation. Moreover, we stop our binary search when the difference between \(\min\) and \(\max\) is at most \(1/n^2\) which is the smallest distance between two average values \((|m/k - m/(k-1)| = m/(k^2 - k) > 1/n^2\) with \(1 < k \leq n\) for \(n\) integers. Thus, the total complexity is \(O(n \log n)\).

**Correctness:** The inequalities above show that we can reduce the data each step by diving in half the search space. When it is impossible to further divide the search space, the solution is given by \((\min + \max)/2\).

### 3. (Description on the Problem Set)

This is a variant of the LIS problem which we can solve in \(O(n \log n)\) (seen in class). We call LDS the longest decreasing subsequence problem. The input is the array \(A\). To solve the bitonic subsequence problem, we simply run the LIS algorithm from left to right where \(LIS[i]\) will contain the length of the longest increasing subsequence ending with \(A[i]\). Similarly the array \(LDS[i]\) will contain the longest decreasing subsequence starting with \(A[i]\) (see this as performing a LIS algorithm on \(A\) from right to left). After that, we can compute the vector \(S[i] = LIS[i] + LDS[i] - 1\). The \(\max_i\{S[i]\}\) will be the maximum length bitonic sequence in \(A\). Note that this approach finds sequences that are increasing and then decreasing. To consider the reverse case, we need to repeat the algorithm by swapping the LIS and the LDS arrays: \(LIS[i]\) will be the length of the longest increasing sequence starting in \(A[i]\) and \(LDS[i]\) will be the length of the shortest decreasing subsequence ending in \(A[i]\).

**Complexity:** Since we perform only additional \(O(n)\) steps, the total complexity is \(O(n \log n)\).

**Correctness:** From the the correctness of LIS and LDS and the fact that in the summation \(S[i] = LIS[i] + LDS[i]\) we are counting the element \(A[i]\) twice, thus we need to subtract 1.
Here is a LIS algorithm that works in $O(n \log n)$. Here we use a set. A set is data structures which maintains all the elements in order after each insertion. It can be implemented as a double-linked list, with an additional pointer to the middle of the list and a smart use of pointers arithmetic (or any binary search tree with $\log n$ worst case, e.g. Red-Black trees). (Note: a similar algorithm works for LDS)

### Algorithm 2 Fast-LIS

1. **procedure** `Fast-LIS(A)` ▷ Input: array $A$
2. Generate an empty set $S$
3. for $0 \leq i < n$ do
4. Insert $A[i]$ into $S$
5. Find the pointer $p$ to element $A[i]$ in $S$
6. Look at element $q$ in $S$ pointed by $p + 1$
7. if $q$ is not the end of $S$ then remove $q$ from $S$
8. end if
9. end for
10. **end procedure** ▷ Solution is size of the set $S$

**Complexity:** Inserting and finding the pointer to an element requires $O(\log n)$, thus the total complexity is $O(n \log n)$.

**Correctness:** If we insert an element $a$ before an existing element $b$ in $S$, then $b > a$ and $b$ was located before $a$ in $A$. Thus, using an induction argument, we can extend the solution by eliminating $b$ and adding $a$, since the remaining optimal solution for $b$ located after the position of $a$ in $A$ is also optimal for $a$.

4. (Description on the Problem Set)

We build a graph $G = (V, E)$ in this way:

- For each cell $c$ in the grid we insert a node $v_c$ in $V$,
- we insert an additional source node $s$ in $V$,
- if two cells $c$ and $c'$ (with heights $h(c)$ and $h(c')$ respectively) are adjacent in the grid we insert an undirected edge $(v_c, v_{c'})$ in $E$, with weight $w(v_c, v_{c'}) = \max(h(c), h(c'))$,
- for each $v_c \in V$ such that $c$ is a cell at the border of the plane surface, we insert the edge $(s, v_c)$ in $E$ with weight $w(s, v_c) = h(v_c)$.

Note that the degree of each node is at most 4. Let $p$ be a path from $s$ to any node $v_c \in V$. We define the height $H(p)$ of the path $p$ as the maximum weight among all the edges on $p$. We want to compute the min-height path from $s$ to a node $v_c$, and let $MH(v_c)$ be the value of such path. Note that $MH(v_c) - h(v_c)$ is the maximum amount of water that can be hold in the cell $c$ after we remove the object from the water. This is true because otherwise the water could flow out from the min-height path and will reach $s$ which represents the empty space around the 3d object (where the water can fall). To compute the $MH(v_c)$ for each node we use a Dijkstra approach where, starting from the source $s$ we relax (propagate) the min-height from each node to the adjacent nodes not yet fully processed.
Algorithm 3 Min Height

1: procedure \( MH(G) \) \quad \triangleright \text{Input: graph } G
2: \hspace{1em} \text{Initialize } MH(v_c) = \infty \text{ for all } v_c \in V \ (MH(s) = 0)
3: \hspace{1em} S = \emptyset
4: \hspace{1em} Q = V
5: \hspace{1em} \text{while } Q \neq \emptyset \text{ do}
6: \hspace{2em} u = \text{Extract-Min}(Q)
7: \hspace{2em} S = S \cup \{u\}
8: \hspace{2em} \text{for each vertex } v \in \Gamma(u) \text{ do}
9: \hspace{3em} \text{if } MH(v) > w(u,v) \text{ then } MH(v) = w(u,v)
10: \hspace{3em} \text{end if}
11: \hspace{2em} \text{end for}
12: \hspace{1em} \text{end while}
13: \hspace{1em} \text{For each } v_c \in V \text{ compute } S(v_c) = MH(v_c) - h(v_c)
14: \hspace{1em} \text{end procedure}

\( H_2O = \sum_{v_c \in V} S(v_c) \)

\textit{Complexity:} Running time of a Dijkstra-like approach is \( O(|E| + |V| \log |V|) \). In this case \( |E| = O(|V|) \) and \( |V| = n^2 \). Thus, the total runtime is \( O(n^2 \log n) \).

\textit{Correctness:} Similar to Dijkstra’s proof, using the invariant method or induction. Moreover, from the above description of the algorithm correctness follows because water does not have any path to escape.