CMSC 27200 - Spring 2024 — Lorenzo Orecchia
due 4/5/24 via Gradescope

H3.

- The assignment is due at Gradescope on 4/5/24.
- A LaTeX template will be provided for each homework. You are strongly encouraged to type your homework into this template using LaTeX. If you are writing by hand, please fill in the solutions in this template, inserting additional sheets as necessary. This will help facilitate the grading.
- You are permitted to discuss the problems with up to 2 other students in the class (per problem); however, *you must write up your own solutions, in your own words*. Do not submit anything you cannot explain. If you do collaborate with any of the other students on any problem, please list all your collaborators in the appropriate spaces.
- Similarly, please list any other source you have used for each problem, including other textbooks or websites.
- *Show your work.* Answers without justification will be given little credit.
- Your homework is *resubmittable*. Please refer to the course syllabus on Canvas for a more detailed description of this. For any problem that you have not changed from your last submission, please make sure to indicate this in your submission to help our graders grade faster.

Problem 1 (Recursive functions) *For the following recursive functions, compute their $O(\cdot)$ complexity:*

1. $T(n) = 7T(n/2) + 4n$
2. $T(n) = 16T(n/4) + n^2$
3. $T(n) = 2T(n/4) + \sqrt{n}$
4. $T(n) = T(n-1) + n^2$

Collaborators: Zack Hassman

Solution:

1. $O(\sum_{i=0}^{log_2 n} (\frac{7}{2})^i n) = O(n^{\log_2 7})$
2. $O(\sum_{i=0}^{log_4 n} (\frac{16}{4^2})^i n^2) = O(n^2 \log n)$
3. $O(\sum_{i=0}^{log_4 n} (\frac{2}{4^{1/2}})^i \sqrt{n}) = O(\sqrt{n} \log n)$
4. $O(\sum_{i=0}^{n} (\frac{1}{4^2})^i n^2) = O(n^3)$

PROBLEM 2 (MAFIA GAME) *A group of $n > 2$ friends are playing a game of Mafia. In each round of this game, the n players can select any subset of the n players to interrogate. Some number of the players are* ***Mafia****, and they will be revealed during the interrogation.*

Denote by $P = \{P_1, P_2, \ldots P_n\}$ the set of n players in the game. We may represent the interrogation as a function I that takes as input any non-empty subset of P and outputs a boolean in $\{\textsc{True}, \textsc{False}\}$ indicating whether the subset contains a Mafia member. More formally, if P_i is Mafia, then $P_i \in P' \implies I(P') = \textsc{True}$.

(a) *Suppose that there is only one Mafia member among the n. Provide a description (in words) of a divide-and-conquer procedure that allows the players to identify the mafia member in $O(\log n)$ interrogations.*

(b) *Provide a proof sketch that this procedure always finds the Mafia.*

(c) *Provide a proof sketch that your procedure from part (a) only takes $O(\log n)$ tests.*

(d) *Provide a formal proof that no procedure can guarantee finding the Mafia member in less than $\lfloor \log_2 n \rfloor$ interrogations.*

(e) *(Bonus - Just to think about, no need to submit anything.) Suppose the Mafia member knows your procedure for picking subsets, and can successfully lie in k interrogations (for some constant k). Obviously, if we just run each interrogation from (a) $k + 1$ times, we'll find the Mafia. That seems redundant though. Is there a way for us to guarantee that we'll identify the Mafia in less than $(k + 1) \log n$ interrogations?*

Suppose instead that there are two Mafia in the group of n players. Furthermore, they can cover each other's tracks in the interrogation: if both are in the same interrogation, then the players won't be able to detect that they are Mafia!

Formally, there exist two players P_i and P_j with $i \neq j$ such that

$$I(P') = \begin{cases} \textsc{False} & \text{If neither } P_i \text{ nor } P_j \text{ are in } P', \\ \textsc{True} & \text{If exactly one of } P_i \text{ or } P_j \text{ are in } P', \\ \textsc{False} & \text{If both } P_i \text{ and } P_j \text{ are in } P'. \end{cases}$$

(f) *Describe a procedure for selecting subsets to interrogate that allows the players to identify both Mafia in $O(\log n)$ interrogations. You may invoke your algorithm from part (a).*

(g) *Sketch a proof that your procedure for part (d) is correct.*

(h) *Sketch a proof that your procedure from part (d) takes $O(\log n)$ interrogations.*

Collaborators: Zack Hassman

Solution:

(a) Repeatedly divide the set of players in two halves. Use one call to I to determine which half set has the Mafia member, and repeat the procedure on that half.

If in the current iteration there is only one player in the set, then that player is the Mafia member.

(b) *Proof.*

We use induction on n, the number of players.

In the base case, if $n = 1$ then the procedure correctly says that one player is the Mafia member

In the inductive case, we assume that the procedure works for any set of $k > 1$ players. Then if given a set of $n = k + 1$ players, the procedure divides P into two halves, which both contain $\leq k$ players. Using I chooses the set containing the Mafia member. Applying the procedure recursively to that set finds the Mafia member.

□

(c) *Proof.*

The procedure's runtime can be expressed as:

$$T(n) = \begin{cases} T(\frac{n}{2}) + O(1) & n > 1 \\ 1 & n = 1 \end{cases}$$

By the Master theorem, the procedure is thus $O(log_2 n)$

□

(d) *Proof.*

Given a set of player P of size n, there are n possible players that may be the Mafia member.

The use of I can prune at most $\lceil \frac{n}{2} \rceil$ possible positions.

Therefore $O(log_2 n)$ comparisons must be made to prune the possible members to 1.

□

(f) Assign each player a unique ID from 1 to n. We represent these IDs in binary.

Then for each bit position i, run I on the set of players with IDs whose ith bit is 1 (ex. the second bit of ID=01000 is 1).

If I returns True, call (a) once on that set, and once on the set of all other players.

If I returns False, continue to the next bit position.

(g) *Proof.*

Because the IDs are unique, the two Mafia members **must** have different digits at some bit position. I is therefore guaranteed to return True at that position as a result of the partitioning process described.

I only returns True if there is one Mafia member. So calling (a) on each partition is guaranteed to find the Mafia members.

□

(h) *Proof.*

It takes $\log n$ bits to uniquely ID all n players.

Therefore there are at most $\log n$ positions to go through. At each position I is called once.

After the Mafia members are separated, (a) is called twice. Recall (a) is $O(\log n)$ as well.

Thus, in total the procedure is $O(\log n)$.

□

Problem 3 (Tilings) *Consider a square $N \times N$ grid where $N = 2^k$ is a power of 2, and imagine placing L-shaped domino pieces of area 3 on this grid. A **tiling** of the grid is a way of placing the pieces so that no two pieces overlap and every single grid cell is covered. In this problem you will be tasked with describing a procedure (an algorithm!) to cover the grid using only these L-shaped pieces. In the input grid, one square is excluded, and doesn't need to be covered by any domino piece.*

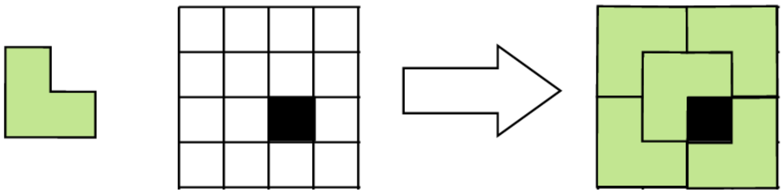


Figure 1: You are given L-shaped domino pieces (in light green) and a grid, and you are asked to find a way to tile the entire grid (excluding the black square). An example of a valid tiling is shown on the right.

(a) *Verify that, if $N = 2^k$, then $N^2 \equiv 1 \bmod 3$, and explain why this condition is necessary (while a priori not sufficient) for a valid tiling to exist.*

(b) *Describe a procedure that allows you to find a valid tiling, given any input grid as above. You may give pseudocode but you are not required to.*

(c) *Sketch a proof that your procedure works correctly.*

Note: every domino piece has the shape described above and covers exactly three squares, every input grid has exactly one excluded (black) square, but the location of the black square may change between input grids.

Collaborators: Zack Hassman

Solution:

(a) $N = 2^k \implies N \equiv_3 4^{k/2} \mod 3 \implies N \equiv_3 (4 \mod 3)^{k/2} \mod 3 \implies N \equiv_3 1$

The dominoes have area 3. The total area of the grid is 2^k. We want to ensure one tile is left over after tiling the grid.

(b) A recursive tiling algorithm:

Algorithm 1 *tile*(k)

```
    Input: $k \in \mathbb{N}$
    Output: Tiled $2^k \times 2^k$ grid
1:  $G \leftarrow 2^k \times 2^k$ grid
2:  if $k = 1$ place tile anywhere in $G$, return $G$
3:  $G_{1/4} \leftarrow tile(k-1)$
4:  Place 3 copies of $G_{1/4}$ in 3 quadrants of $G$ such that their empty squares are in the grid center
5:  Place a copy of $G_{1/4}$ in $G$ such that its empty square is on the grid corner
6:  Fill the 3 empty squares in the middle with a domino
7:  return $G$
```

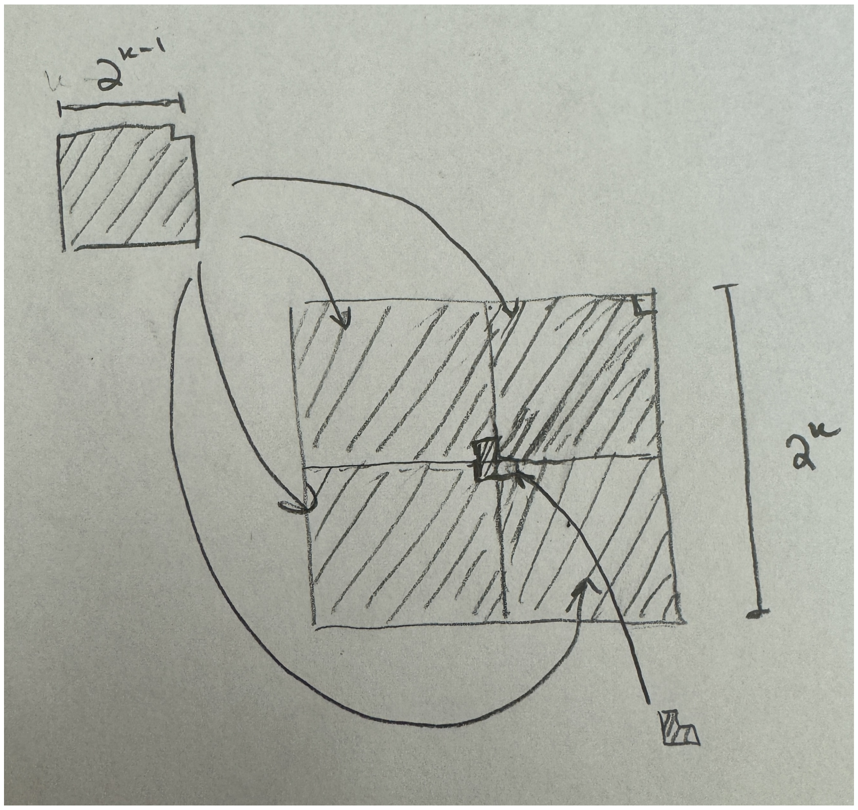
(c) *Proof.*

We use induction on k.

In the base case, if $k = 1$, any placing of the domino is a valid tiling, and leaves a grid corner empty. The procedure does this.

In the inductive case, we assume that the procedure can correctly tile a $k - 1$ grid leaving an empty corner. Then in the case of a k grid, each quadrant is a $k - 1$ grid. The procedure can compute the tiling of the $k - 1$ grid by the inductive hypothesis. We can arrange 3 such tilings such that all the empty corners are in the middle of the k grid. Then we can fill these empty squares with one domino. One final $k - 1$ grid can be put in the remaining quadrant to complete the k grid tiling.

□



Problem 4 (Benchmarking Insertion Sort) *After reading the correctness proof of insertion sort during discussion for week 1, Konstantinos is wondering how many times line 6 is performed for a given array. He could simply count, but it took too long for big arrays, so instead he wants you to write a faster algorithm to compute that result.*

Algorithm 2 Insertion Sort

```
    Input: Array $A$
    Output: Array $A$
1:  for $j = 2$ to $A$.length do
2:      $key = A[j]$
3:      // Insert $A[j]$ into the sorted sequence $A[1..j-1]$.
4:      $i = j - 1$
5:      while $i > 0$ and $A[i] > key$ do
6:          $A[i+1] = A[i]$
7:          $i = i - 1$
8:      end while
9:      $A[i+1] = key$
10: end for
11: Return $A$
```

Input: *The first line contains a single integer N, the size of the array. In the next line there are N numbers, from* 1 *to N.*

Output: *A single number, how many times line 6 of insertion sort would be executed on this array.*

Submission: *In Gradescope upload a single Python file. Skeleton code to get you started can be found on Canvas.*

Testing: *Test case inputs and outputs are provided here. To compare your output to the given output you can use the following command on Linux and MacOS Terminal*

```
pypy3 benchmarking_insertion.py <testcases/input01.txt | diff testcases/output01.txt -
```

Limits:
$N \leq 1.000.000$
$T \leq 2s$

Sample Input	Sample Output
5 2 5 3 4 1	6