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:

Solution: We can compuite complexity by Master Theorem as we have the functions in the form of $aT(\frac{n}{b}) + O(n^d)$.

1. $T(n) = 7T(n/2) + 4n$: We have $a = 7$, $b = 2$ and $d = 1$. As $7 > 2^1$, we have $T(n) = O(n^{\log_2 7})$.
2. $T(n) = 16T(n/4) + n^2$: We have $a = 16$, $b = 4$ and $d = 2$. As $16 = 4^2$, we have $T(n) = O(n^2 \log n)$.
3. $T(n) = 2T(n/4) + \sqrt{n}$: We have $a = 2$, $b = 4$ and $d = 0.5$. As $2 = \sqrt{4}$, we have $T(n) = O(\sqrt{n} \log n)$.
4. $T(n) = T(n-1) + n^2$: Master Theorem is not applicable, but this recursive function can be transformed to $T(n) = T(n-1) + n^2 = (T(n-2) + T(n-1)^2) + n^2 = \cdots = T(1) + 2^2 + \cdots + n^2$ which can be computed as $\frac{n(n-1)(2n+1)}{6}$, which has $O(n^3)$.

Extra space for your solution

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 $\{\text{True}, \text{False}\}$ indicating whether the subset contains a Mafia member. More formally, if P_i is Mafia, then $P_i \in P' \implies I(P') = \text{True}$.

(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

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

(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:

Solution:

(a) Split the n players into two roughly equal groups, each of size $\frac{n}{2}$. Interrogate both groups. If group A contain a Mafia member, then the Mafia member is in group A. Otherwise, the Mafia member is in group B. Without losing generality, we can assume that the Mafia member is in group A. Repeat the process with group A to divide two groups and interrogate them. Continue this process until we have only one player left, which is the Mafia member. This process takes $O(\log n)$ interrogations as each step halves the group size.

(b) At each step, we are able to halve the group size, and one of these subgroups is eliminated based on the interrogation result. This process is repeated until we have only one player left, which is the Mafia member. As we assume there is only one Mafia member, and since the Mafia member cannot be eliminated in any interrogation, we can guarantee that our procedure eventually isolate the Mafia member.

(c) Each interrogation will halve the group size. Starting with n players, after the first interrogation we have $n/2$ players, after the second interrogation we have $n/4$ players, and so on. After k interrogations, we have $n/2^k$ players. The worst case to find the Mafia number is the $n/2^k$th player is the Mafia, so we have $n/2^k = 1$. This gives us $k = \log_2 n$, and therefore $O(\log n)$.

(d) We can prove this by contradiction. Suppose there is a procedure that can find the Mafia member in less than $\lfloor \log_2 n \rfloor$ interrogations. This means that the procedure can find the Mafia member in k interrogations, where $k < \lfloor \log_2 n \rfloor$ as we don't want players to remain after k interrogations. Therefore, the procedure can find the Mafia member in $2^k < n$ players. However, since there is only one Mafia member, and the Mafia member can be in any of the n players, we cannot guarantee to find the Mafia member in less than $\lfloor \log_2 n \rfloor$ interrogations.

(e) (Thoughts) Introduce more reduncancy and see the pattern?

(f) Assign each player a unique number from 0 to $n-1$ in binary representation. For each bit, we divide the players into two groups based on the bit value. For each bit position, if the interrogation returns True, we record the bit position and move on to the next bit position until Mafia members can be distinctly identified.

(g) Since each player has a unique binary identifier, there is guaranteed to be at least one bit position that differentiates any two players, including the two Mafia members. By iterating through all bit positions and dividing players based on these, you eventually find bit positions where the Mafia members are separated into different groups

(h) As we need to iterate through all bit positions, the number of bits needed to identify all players is $\log_2 n$. For each bit position, we divide the players into two groups and interrogate both. Therefore we need $2\log_2 n$ interrogations, which gives us $O(\log n)$.

Extra space for your solution

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.

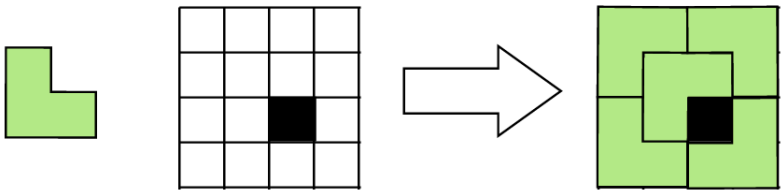

Figure 1: You are given L-shaped domino pieces (in light green) and a grid, and you are asked to find a way to tile the entire grid (excluding the black square). An example of a valid tiling is shown on the right.

(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:

Solution:

(a) $N^2 = (2^k)^2 = 2^{2k}$ As we have $2^2 \equiv 1 \bmod 3$, we can write 2^{2k} as $(2^2)^k \equiv 1^k \equiv 1 \bmod 3$. This condition is necessary as each domino piece covers 3 squares, and the grid has N^2 squares. To cover the entire grid except for one square, the total number of squares must be one more than a multiple of 3, which must be ensured by $N^2 \equiv 1 \bmod 3$.

(b) We can use a recursive algorithm to find a valid tiling. The algorithm can be described as follows:

(1) Divide the grid into four subgrids of size $\frac{N}{2} \times \frac{N}{2}$ and place the domino piece at the center such that it does not cover the square that has a missing square. This makes all four subgrids have a missing square.

(2) Recursively apply the algorithm to each of the four subgrids until all squares are filled out.

(c) We can prove the correctness by induction: Assume the tiling procedure works correctly for a grid of size $2^k \times 2^k$ with one square excluded. We will prove it works for a grid of $2^{k+1} \times 2^{k+1}$.
Base case: For $k = 1$, the grid is 2×2, and the tiling works.
Inductive step: We can divide the grid into four subgrids of size $2^k \times 2^k$, and place the domino piece at the center such that it does not cover the square that has a missing square. This makes all four subgrids have a missing square. By applying the algorithm recursively to each of the four subgrids, we can fill out all squares. By tiling each subgrid and having the central domino in place, we achieve successful tiling for each 2^k grid, and therefore the tiling works for $2^{k+1} \times 2^{k+1}$ grid.

Extra space for your solution

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 1 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