CSE 101
HW2 Review Session
Week 3

January 22 - January 29
Important

Homework solutions:

▶ Any solution to a problem must include (if not stated otherwise) high-level description, pseudo-code, formal proof of time complexity and formal proof of correctness. See solution templates.

▶ Use of google search, chegg, or any other resource that is NOT explicitly permitted is a violation of A.I. rules and will be reported per UCSD / JSOE / CSE policy.

▶ Any use of ALLOWED sources (e.g., discussions in study group) must be properly cited.

▶ Please refer to model solutions at the end of the website for examples of style.

▶ Please study HW1 solutions for additional style examples.
Problem 1 (DPV 2.5)

Let for some $a \geq 1$ and $b > 1$:

\[
T(1) = \Theta(1)
\]

\[
T(n) = aT\left(\frac{n}{b}\right) + \Theta(n^d) \text{ for } n > 1
\]

Then:

- If $a < b^d$, then $T(n) = \Theta(n^d)$
- If $a = b^d$, then $T(n) = \Theta(n^d \log(n))$
- If $a > b^d$, then $T(n) = \Theta(n^{\log_b(a)})$

In the lectures you had $O$ instead of $\Theta$. Both variations are true.
Problem 1 (DPV 2.5)

1. \( T(n) = 2T\left(\frac{n}{2}\right) + n^2 \)
   \( a = 2, b = 2, d = 2, a < b^d \), thus, \( T(n) = \Theta(n^2) \)

2. \( T(n) = 6T\left(\frac{n}{2}\right) + n^3 \)
   \( a = 6, b = 2, d = 3, a = b^d \), thus, \( T(n) = \Theta(n) \)

3. \( T(n) = 3T\left(\frac{n}{2}\right) + n \)
   \( a = 3, b = 2, d = 1, a > b^d \), thus, \( T(n) = \Theta(n^{\log_2(3)}) \)

4. \( T(n) = T(n/2) + \log(n) \)
   \( \log(n) = O(n) \) and \( a = 1, b = 2, d = 1, a < b^d \), thus,
   \( T(n) = O(n) \), but we can’t say anything about the \( \Theta \) bound.
Problem 1 (DPV 2.5)

- Unraveling
- Guess and check
- Constructive induction (covered in this week’s discussion sections)

You are only required to find the big $\Theta$ bound.

Base case: $T(x) = 0$ for any $0 \leq x < 1$. 
Problem 2 (DPV 2.22)

- It is easy to come up with the solution that runs in $O(n + m)$: merge two sorted arrays and choose the $k$th element.
- Can be solved faster with Divide-and-Conquer.
- Problem size is defined by $(n, m)$. Can it be divided into smaller subproblems?
- Note that the required complexity is $O(\log(n) + \log(m))$. Logarithm usually means dividing the size of a problem by two.
- Consider elements in the middle of both arrays – $A[n/2]$, $B[m/2]$ and compare them. Depending on the result and some additional information, where can the $k$th element be located? Can we get rid of some parts of one (or maybe both) of the arrays and get the subproblem of the same type, but of smaller dimensions?
- Many solutions exist: $O(n + m)$, $O(\log(n) + \log(m))$, $O(\log(n + m))$, and $O(\log(\min(n, m)))$. 
Problem 3 (DPV 2.12, 2.14, 2.16)

(a) Examples from the first week’s discussion section may be helpful (see the slides)

(b) ▶ Preprocessing may be useful. Think about the order of the elements.
    ▶ Or think about using Divide-and-Conquer. If you removed duplicates in smaller subarrays, how can you combine the results and remove all the duplicates in the initial array? Think again about the order of the elements.

(c) We can’t run a binary search in an infinite array. We need to find boundaries $l, r$, such that $A[l] \leq x \leq A[r]$ and $l \leq r$. We can use binary search in the subarray $A[l...r]$. How to find boundaries $l, r$ in an efficient way? ($O(\log(n))$)
Problem 4 (DPV 2.27)

(a) Express the elements of matrix $A^2$ in terms of elements of $A$ and try to minimize the number of multiplications (do not multiply the same elements twice by storing the product and use simple arithmetic)

(b) Does the updated algorithm still follow Divide-and-Conquer paradigm? What about its time complexity? (a recurrent relation)

(c) Imagine you have a function $f(X) = X^2$, which finds the square of some matrix $X$. This function runs in $S(n) = \mathcal{O}(n^c)$. You have this function and you can perform addition and subtraction, because they only take $\mathcal{O}(n^2)$. How do you find $AB + BA$ in $3S(n) + \mathcal{O}(n^2)$?
The graph is undirected and has unit length weights. Which algorithm is the most efficient for finding the shortest distances in such graphs?

Let \( d(x, y) \) be the shortest distance between nodes \( x, y \) in \( V \). Then we need to find the number of paths of length \( d(u, v) \) from \( u \) to \( v \).

Let \( x \) be some node that is reachable from \( u \) and adjacent to \( v \). How is the number of paths from \( u \) to \( v \) of length \( d(u, v) \) related to the number of paths from \( u \) to \( x \) of length \( d(u, x) \)?

One of the possibilities: make updates that take constant time to existing algorithm(s) that run in linear time.
The graph has positive weights. Which algorithm can be applied to find the shortest distance from one node to all others?

The graph is strongly connected, thus for any pair of nodes \((u, v)\), there is a path from \(u\) to \(v\) and there is a path from \(v\) to \(u\).

Examples from the third week’s discussion section may be helpful (see the slides).
Problem 7 (DPV 4.19)

You may use one of the methods to approach the problem:

1. Make updates to the graph. You may construct another graph (with different number of nodes and edges possibly), such that finding the shortest distance in the new graph between some nodes is equivalent to finding the shortest distance in the initial graph between corresponding nodes.

2. Make updates to the algorithm. Update Dijkstra’s algorithm, such that it can find the shortest distances in the given graph. The overall time complexity should not change.