Homework 1

Due: 10 October, 2025 6pm PT

Unless stated otherwise, you should always prove the correctness of your algorithms and analyze their runtimes.

Each problem is graded on the coarse scale of ✓$^+$, ✓, ✓$^-$ and no ✓. It is also assigned a multiplier, denoting the relative importance of the problem. Both correctness and presentation are grading criteria.

Please read and make sure you understand the collaboration policy on the course missive. Extra credit problems are clearly marked below (see course missive for the details of grade calculations).

Remember to prove all your (non-elementary and not shown in class) mathematical claims, unless stated otherwise.

Each group of students should submit only 1 pdf to the corresponding Gradescope assignment.

Problem 1

(5 ✓s)

Use induction to prove the *correctness* of these following simple algorithms (presented as code below). Re-read the posted handwritten notes on Lecture 1 to see how Jasper talks about algorithm correctness.

1. (2 ✓s) Show that the following function computes the sum of an input array A.

   SUM(array A)

   1 $n \leftarrow A.length$
   2 $out \leftarrow 0$
   3 for $i \leftarrow 1$ to n: $out \leftarrow out + A[i]$
   4 return out

2. (3 ✓s) Show that the following (recursive) function implements search in a complete binary search tree T: given the tree T and a number x being searched for, the function returns "yes" if x is in T and "no" otherwise. Here, a complete binary tree is either an empty tree (think, leaf node), or any internal node must have 2 children (node.l and node.r) and a value (node.val). As a binary *search* tree, for any internal node, node.val is strictly bigger than every value in node.l, and strictly smaller than every value in node.r.

   SEARCH(Complete binary search tree T, number x)

   1 if T is empty: return "no"
   2 elseif $T.val = x$: return "yes"
   3 elseif $T.val > x$: return SEARCH($T.l$, x)
   4 elseif $T.val < x$: return SEARCH($T.r$, x)

Tips for Problem 1

Make sure you have a firm understanding of why an algorithm works before you analyze its correctness. Then, try to figure out what variable to induct on: for loops, it is frequently "time" or "loop index" that is the variable being inducted on; for recursive code, it is frequently "size of input" that is being inducted on. A good rule of thumb (and a necessary thing to do) is to induct on a variable that is *changing* throughout the computation process.

The next step is to clearly state the induction hypothesis: what must remain true as the induction variable changes throughout the computation? Be sure to be mathematically precise, and don't just use vague words like "correct". What does "correct" actually mean in your context?

With the induction hypothesis we can now get into the weeds of actually proving correctness. Proving the base case should (usually) be simple, and ideally not very interesting. Proving the induction step is (usually) the meat and the hardest part of the proof. Why does the induction hypothesis **continue to hold** as the induction variable changes? Be sure to invoke the induction hypothesis on smaller/previous induction variable values when proving the induction step; otherwise something is definitely wrong.

To conclude the induction, write a brief sentence (or two, or more if necessary) reminding the reader what the induction has accomplished.

Problem 2

(2 ✓s)

There is a long row of houses numbered $1, 2, 3, \ldots$, and Gary is throwing a party at house number h; but sadly, you do not know what h is. You can go to any house i and ask them where the party is, and they will tell you either "keep going" if $i < h$, or "you passed it" if $i > h$, or "Party's here!" if $i = h$. Give an algorithm to find the party, using $O(\log h)$ queries.

(**Hint**: right now we don't know anything about h. But let's say we know that $h \in [1..n]$, for some parameter n; what would you do? After that, how would you efficiently find a parameter n that isn't too big? The last part is the main challenge, and a nice trick that you should learn.)

Problem 3

(3 ✓s)

Prove via divide-and-conquer that, given a $2^k \times 2^k$ board of squares with one square removed, you can exactly cover the entire board with L-shaped pieces (each covering exactly 3 squares).

Problem 4

(**Extra Credit**, 2 ✓s)

Alice told Bob about an interesting recursive class of matrices, but unfortunately forgot to tell Bob how to efficiently multiply them with vectors. Alice's matrices are of sizes that are powers of 2 and are defined recursively as follows:

$$E_1 = \begin{pmatrix} 1 \end{pmatrix}$$

$$E_2 = \begin{pmatrix} 1 & -1 \\ 3 & 5 \end{pmatrix}$$

$$E_4 = \begin{pmatrix} 1 & -1 & -1 & 1 \\ 3 & 5 & -3 & -5 \\ 3 & -3 & 5 & -5 \\ 9 & 15 & 15 & 25 \end{pmatrix}$$

$$\vdots$$

$$E_{2^k} = \left(\begin{array}{c|c} 1 \cdot E_{2^{k-1}} & -1 \cdot E_{2^{k-1}} \\ \hline 3 \cdot E_{2^{k-1}} & 5 \cdot E_{2^{k-1}} \end{array}\right)$$

Find an algorithm that, when n is a power of 2, lets Bob multiply E_n by length n vectors in $O(n \log n)$ time. This is much faster than the $O(n^2)$ time it would take to compute the product using standard linear algebra. (**Note:** Do *not* construct the matrices E in your code, as even constructing these matrices will take $O(n^2)$ time, which is too much!)

(This problem is a much simpler version of the idea underlying the Fast Fourier Transform.)

Problem 5

(**Extra Credit**, 3 ✓s)

Suppose we have an $n \times n$ matrix (grid) of distinct integers. A local minimum is a location (i, j) whose entry is smaller than all its neighbors (up, down, left, right, or the applicable subset of directions along the edges). Find an algorithm that finds a local minimum using only $O(n)$ comparison queries between pairs of locations.

(**Hint**: Try recursing into one of the $n/2 \times n/2$ quadrants after only $O(n)$ queries. To decide on which quadrant to use, find the minimum across an $O(n)$ number of locations. You will want the set of locations considered to include at least one full row and one full column.)