CS302 - Problem Set 1

Please familiarize yourself with the sections of the syllabus on the honor code and problem sets before starting this homework.

Solutions and hints for several parts of this problem set are on the final page so you can check that you are on the right track or get moving if you are stuck, but try to do as much as possible on your own first. To make the assignment more challenging, do not look at the hints.

1. This problem will help you to review and recall logarithms, exponentiation, summation, trees, recurrence relations, geometric series, and big-O notation. If you would like additional review, see

   - CS200 texts
   - Log Review resources
   - Big-O text

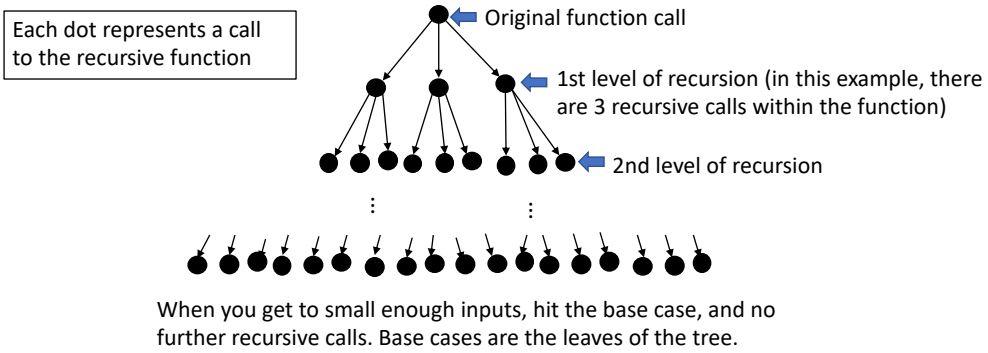
   Let $T(n)$ be the runtime of a recursive algorithm on input size n. $T(n)$ is described by the recurrence relation

   $$T(1) = O(1), \qquad T(n) = aT(n/b) + O(n^d), \tag{1}$$

   where a, b, and d are constants.

   (a) What do a, b, and d represent in terms of the structure of the algorithm?

   (b) We can describe the structure of a recursive algorithm with runtime $T(n) = aT(n/b) + O(n^d), T(1) = O(1)$ as a tree:



   Each call to the function is represented by a vertex, the initial call of the function is the root, and each recursive call produces a child vertex from the vertex of the function that called it. In terms of n (the original input size), a, b, and/or d, how many levels will this tree have (from the root to the leaves)? In other words, how many levels of recursion will there be?

(c) In terms of n (the original input size), a, b, and/or d, how many function calls are there at the kth level of recursion? In terms of the tree, we can rephrase this question as how many vertices are there in level k below the root?

(d) At each level of recursion, the size of the input to the function decreases. If the original call had an input of size n, in terms of n, a, b, and/or d, what is the size of the input to the function calls at level k?

(e) In terms of n (the original input size), a, b, and/or d, at a single vertex (function call) at level k, how much time (how many operations) is used by that function call, excluding any operations done in the recursive calls it makes. For example, in MergeSort, if the input to a function call is size m, the work done at that call and ignoring the two recursive calls is $O(m)$, because we only look at the for loop part of the algorithm, which takes time $O(m)$.

(f) In terms of n (the original input size), a, b, and/or d, how much time (how many operations) is used by all function calls at level k, excluding any operations done by the further recursive calls they make? (Combine your answers to (c) and (e).)

(g) In terms of n (the original input size), a, b, and/or d, how much time (how many operations) is used by all function calls in the algorithm (at all levels of recursion)? (Please leave in summation notation.)

(h) Use the formula for geometric series:

$$\sum_{k=0}^{t} r^k = \begin{cases} t+1 & \text{if } r = 1 \\ \frac{r^{t+1}-1}{r-1} & \text{else} \end{cases} \tag{2}$$

to evaluate the sum from the previous question.

(i) If $\frac{a}{b^d} = 1$, what is the big-O runtime of the algorithm? You should assume a, b, and d are constants, and n is the variable that gets large.

(j) If $\frac{a}{b^d} < 1$ what is the big-O runtime of the algorithm? You should assume a, b, and d are constants, and n is the variable that gets large.

(k) If $\frac{a}{b^d} > 1$ what is the big-O runtime of the algorithm? You should assume a, b, and d are constants, and n is the variable that gets large.

(l) You should find that the runtime behaves differently depending on the 3 possible relationships between $\frac{a}{b^d}$ and 1. Qualitatively explain the behavior in each case. I'll do one case as an example: if $\frac{a}{b^d} < 1$, that means that a tends to be small relative to b and d. If b and d are large, that means that recursive calls happen on inputs that are much smaller than the original input, and a lot of time/operations are spent *not* in recursive calls. If a is small, that means there are not a lot of recursive calls. Putting these ideas together, we expect that in this case the runtime will *not* be strongly dependent on the recursive calls. From our analysis in part (j), we see that when $\frac{a}{b^d} < 1$, the whole runtime of the algorithm is $O(n^d)$. However, note that the original function call uses $O(n^d)$ time, excluding recursive calls! This means that all of the recursive calls in the whole algorithm are basically not adding anything significant to the runtime, and the majority of the runtime is spent at the top of the tree, at the root. This makes sense, given our qualitative explanation that we expect the runtime to not be strongly affected

by recursive calls. (Now you should do a similar analysis for the other two cases: $\frac{a}{b^d} > 1$ and $\frac{a}{b^d} = 1$.)

2. The elements of a bi-tonic list either only increase, only decrease, or first increase and then decrease.

   (a) Write pseudocode for a recursive algorithm `BMax` that finds the maximum value of a bi-tonic list of n distinct integers that runs in time $O(\log n)$.

   Algorithm 1: `BMax`(A)

   Input : Bi-tonic list A of distinct integers of size n
   Output: The maximum value of A)
   `// Your pseudocode here!`

   (b) Prove your algorithm is correct. In the hints, I've given you two strong induction templates to use if you would like some help getting started.

   (c) Create a recurrence relation for the runtime of your algorithm.

   (d) Evaluate the recurrence relation using your result from problem 1, showing that the algorithm indeed has runtime $O(\log n)$, and also provide an intuitive explanation for why the runtime is what it is.

   (e) (Challenge) How would each part of the problem change if we additionally allow lists that first decrease and then increase? What about if we allow lists that change direction twice (first increase, then decrease, then increase; or first decrease, then increase, then decrease) or change direction k times?

3. For the closest points problem, we argued we only needed to look at points with x-coordinates that were within $\pm\delta$ of the midline, where δ is the smallest separation found in either the left half or the right half of points. In this problem, we consider how that analysis would change if we were to calculate the distance between points differently. Decide what range of x-coordinates away from the midline you should consider if we instead used

   (a) The Minkowski distance: $d(p_i, p_j) = (|x_i - x_j|^p + |y_i - y_j|^p)^{1/p}$, for $p \in \mathbb{R}^+$. This distance is what you get on curved spacetime like in general relativity.

   (b) The skewed distance: $d(p_i, p_j) = \sqrt{2(x_i - x_j)^2 + (y_i - y_j)^2}$. This distance would make sense in a scenario where it is much harder to travel in the x-direction than in the y-direction. For example, suppose to travel in the y-direction, you can get on highways, but in the x-direction, you have to take local roads...like in Vermont! (Just try to get to Maine from here!)

Hints:

1.
   - The following are a bunch of "tools" that will be helpful:
     - $x = y^{\log_y x}$
     - $x^{\log_y z} = z^{\log_y x}$
     - $(x^y)^z = (x^z)^y = x^{yz} = x^{zy}$
     - $c^{y+1} = c \times c^y$
     - If c is a constant, $O(\log_c n) = O(\log n)$
     - If $f(x) \geq g(x)$, then $O(f(x) + g(x)) = O(f(x))$.
     - If $0 < r < 1$, then $0 < r^t < 1$ for any power t greater than 0.
   - The solution to part (g) is

$$T(n) = \sum_{k=0}^{\log_b n} a^k \left(n/b^k\right)^d. \tag{3}$$

     But if we pull out n^d and group terms that are raised to the power of k together, we get

$$T(n) = n^d \sum_{k=0}^{\log_b n} \left(\frac{a}{b^d}\right)^k \tag{4}$$

   - The solution to part (i-j) is

$$T(1) = O(1)$$
$$T(n) = \begin{cases} n^d \log_b n & \text{if } a = b^d \\ n^d & \text{if } a < b^d \\ n^{\log_b a} & \text{if } a > b^d \end{cases} \tag{5}$$

2.
   - For part (a), your algorithm should look very similar to binary search.
   - For part (b), here are two templates:

     Proof. We will prove `BMax` correctly returns the maximum value of any bi-tonic array of size $n \geq 1$ using strong induction.
     Base case: When $n = 1$, ...
     Inductive step: We assume `BMax` correctly returns the maximum value of any bi-tonic array of size at most k, for $k \geq 1$. Now consider an input of size $k + 1$... □

     Proof. Let $P(n)$ be the predicate `BMax` correctly returns the maximum value of any bi-tonic array of size n. We will prove $P(n)$ is true for all $n \geq 1$.
     Base case: $P(1)$ is true because...
     Inductive step: Assume $P(j)$ is true for all j such that $1 \leq j \leq k$. Now consider an input of size $k + 1$... □