# CS302 - Problem Set 1

1. Please read over the syllabus at go/cs302 (you do not need to follow any links at this time, unless you are interested - you can follow up later as needed), and post at least one question, concern, or comment at this Canvas Discussion. Please also "like" any other posts you agree with.

2. Many of you took the CRA computing survey in a CS class at the end of the Fall semester. If you did not take it in Fall 2021, please do so now: survey link. (If you took it last academic year 2020-2021, but not yet this academic year, you should take it again.)

3. The following are videos/resources to fill in or refresh material that is helpful for solving the rest of the problem set (and are topics that you were very likely exposed to in 200/201, and that we will continue to practice throughout the semester). You are not required to watch/read them, but if you are getting stuck as you work you on the problem set, you should check out the relevant topic.

   - Video: Runtime Recurrence Relation 17 min; steps through creating a recurrence relation for the runtime of a recursive algorithm.
   - Video: Induction to Prove Correctness of Recursive Algorithms 25 min; Discusses (with examples) proving the correctness of recursive algorithms using induction.
   - Video Expand and Hope Method 12:42; Gives an example of the Expand and Hope method for solving recurrences.
   - Log Review resources
   - Textbook Chapter on Big-O

4. The runtime of many recursive algorithms takes the following form:

$$T(n) = \begin{cases} O(1) \text{ if } n \leq c, \text{ for } c = O(1) \text{ (base case)} \\ aT(n/b) + O(n^d) \text{ otherwise (recursive relation)} \end{cases} \tag{1}$$

   where $T(n)$ is the runtime of the function on an input of size $n$, and $a$, $b$, $c$, and $d$ are constants and that do not depend on $n$. When the runtime has this recursion relation, then you can use the following formula (which I'll call the *tree formula*) to determine the overall runtime:

$$T(n) = \begin{cases} O(n^d \log_b n) & \text{if } a = b^d \\ O(n^d) & \text{if } a < b^d \\ O(n^{\log_b a}) & \text{if } a > b^d \end{cases} \tag{2}$$

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

   (b) Each of the following algorithms represents a variation on searching an ordered list. For each one:

- Derive a recurrence relation for the runtime.
- If you can use the tree formula, do so to determine the runtime. If you can't use the tree formula, explain why not, and instead use the *expand and hope* method (see video, also called the *expand and pray* method, or the *iterative* method).

i.

**Algorithm 1:** `Search1`$(A, v)$

```
Input  : Sorted list $A$ of distinct integers of size $n$, a value $v \in A$
Output: The index $i$ such that $A[i] = v$
// Base Case
1  if $n \leq 10$ then
2      for $i \leftarrow 1$ to $n$ do
           // I always start with index 1 in pseudocode.
3          if $A[i] = v$ then
4              return $i$;
5          end
6      end
       // If target not found:
7      return $-1$;
8  end
// Recursive Step
9  mid1← $\lfloor n/3 \rfloor$;
10 mid2← $\lfloor 2n/3 \rfloor$;
11 return max{Search1($A$[1 :mid2], $v$),Search1($A$[mid1: end], $v$)};
// $A[s:f]$ denotes the subarray from index $s$ to $f$ inclusive
```

ii.

**Algorithm 2:** `Search2`$(A, v, s, f)$

```
Input  : Sorted list $A$ of distinct integers of size $n$, a value $v \in A$, indeces $s$, $f$ to demarcate
         endpoints of subarray
Output: The index $i$ such that $A[i] = v$
// Base Case
if $f - s \leq 1$ then
    return s;
end
// Caffeinate
for $j = s$ to $f$ do
    Take a coffee break;
end
// Check if endpoint has the target value
if $A[f]{=}v$ then
    return f;
end
// Recursive step
return Search2$(A[s : f-1], v, s, f-1)$;
```

iii.

**Algorithm 3:** `Search3`$(A, v, s, f)$

```
Input  : Sorted list $A$ of distinct integers of size $n$, a value $v \in A$, indeces $s$, $f$ to demarcate
         endpoints of subarray
Output: The index $i$ such that $A[i] = v$
// Base Case
if $f - s + 1 \leq \sqrt{n}$ then
    for $i = s$ to $f$ do
        if $A[i] = v$ then
            return i;
        end
    end
end
// Check if midpoint is target value
mid=$\lfloor (f + s)/2 \rfloor$;
if $A[mid]=v$ then
    return mid;
else
    // Recursive step
    if $A[mid] > v$ then
        return Search3$(A, v, s, mid - 1)$;
    else
        return Search3$(A, v, mid + 1, f)$;
    end
end
```

(c) (Challenge) One of the algorithms has an error in it and will not return the correct value on certain inputs. (It's possible there are other errors that I haven't found yet!) Which one is it and why?

5. The elements of a bi-tonic list either only increase, only decrease, or first increase and then decrease. For example: $[1, 4, 5, 10, 14]$, $[1, 8, 5, 4]$, $[1, 0]$, and $[3]$ are all bitonic, while $[5, 4, 3, 4, 5]$ and $[3, 5, 3, 5]$ are not bitonic.

   (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 4:** `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 video, I use a more formal inductive proof style with a predicate $P(n)$. You don't have to use a predicate if you prefer to set up your inductive proof less formally, as long as the structure and logic are still there.)

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

   (d) Evaluate the recurrence relation using the tree formula, 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?