CSC373 Summer '22
Assignment 1

Due Date: June 2, 2022, 11:59pm ET

Instructions

1. Typed assignments are preferred (e.g., PDFs created using LaTeX or Word), especially if your handwriting is possibly illegible or if you do not have access to a good quality scanner. Either way, you need to submit a single PDF named "hwk1.pdf" on MarkUS at `https://markus.teach.cs.toronto.edu/csc373-2021-09`
2. You will receive 20% of the points for a (sub)question when you leave it blank (or cross off any written solution) and write "I do not know how to approach this problem." If you leave it blank but do not write this or a similar statement, you will receive 10%. This does not apply to any bonus (sub)questions.
3. You may receive partial credit for the work that is clearly on the right track. But if your answer is largely irrelevant, you will receive 0 points.

Q1 [20 Points] Decide the Candidate(s)!

Imagine you are the election officer of a residential township with n eligible voters. There has already been a round of preliminary voting in which each voter has given you the name of *one* person who they would like to see run as a candidate in the upcoming election. As a result, you have a list of n names (a name can, of course, repeat multiple times in this list – especially the popular ones!). Any person who receives (strictly) more than $n/3$ votes in the preliminary voting is to be declared a candidate. Note that hence, there can only be $0, 1$, or 2 candidates as a result of preliminary voting. Your task as the election officer is to figure out if there even is someone who is to be declared a candidate, and if so, who the candidate(s) is/are.
Assume that you are *only* allowed to check two strings for equality (i.e., if two given names on the list are the same) – which can be done in constant time. In particular, assume that there is no "ordering" possible on the strings (i.e., for example checking which of some two strings comes first in an alphabetical order is impossible).
Note that the brute-force way to do this takes $O(n^2)$ time: for every name on the list, you run through the entire list, count how many times it appears on the list, and check if the count is strictly larger than $n/3$.

(a) [10 Points] Give a divide-and-conquer algorithm that solves this problem in $O(n \log n)$ time.

(b) [5 Points] Analyze the running time of your algorithm (and provide the recurrence relation).

(c) [5 Points] Briefly justify why your algorithm is correct.

Solution to Q1

(a) Let $A[1, \ldots, n]$ be the list of the given n names. The basic idea is that any element that appears in more than $n/3$ positions of A must appear in more than $n/6$ positions of $A[1, \ldots, \lfloor n/2 \rfloor]$

or $A[\lfloor n/2 \rfloor, \ldots, n]$, or both – otherwise, it wouldn't appear in $n/3$ positions overall. The algorithm finds all such elements (up to two of them) recursively for each half of the list – any element that appears in more than $n/6$ positions of *both* halves automatically appears in more than $n/3$ positions overall; for other such elements, simply perform a linear-time search of the list to determine their number of occurrences overall.

Algorithm 1 Algorithm to determine the candidate(s)

```
procedure CANDIDATES(A)
    n ← length(A)
    if length(A) ≤ 2 then
        # Return all elements in A, removing duplicates.
        return {A[1], ..., A[n]} # as a set, i.e., no duplicates
    else
        # Find candidate elements in each half.
        F ← CANDIDATES(A[1, ..., ⌊n/2⌋])
        S ← CANDIDATES(A[⌊n/2⌋ + 1, ..., n])
        # If a name is a candidate in both halves, it means it is a candidate overall
        # for other names, count occurrences
        Let M ← F ∩ S
        for e ∈ (F ∪ S) \ M do
            if the number of occurrences of e in A is > n/3 then
                M ← M ∪ {e}
            end if
        end for
        return M
    end if
end procedure
```

(b) The runtime $T(n)$ satisfies $T(n) = 2T(n/2) + O(n)$. This is because after computing F and S, we compute a linear search for the elements in $(F \cup S) \setminus M$, which are only constantly many. So by Master Theorem, $T(n) = O(n \log n)$, as desired.

(c) The proof of correctness is by induction: the base case when the length of the list is smaller than 3 is clear – all elements are candidates! Otherwise, for larger n, after obtaining the candidates F, S from either halves, the proof of correctness follows from the basic observation that was made at the beginning of part (a): any overall candidate must be a candidate in at least one of the halves, and so a linear search for these constantly many elements suffices.

Q2 [20 Points] Treasure hunt!

A treasure hunter has a map of n treasures. For each treasure i, the treasure hunter assigns two integer valued parameters, namely the easiness x_i to obtain the treasure, and its value y_i. The treasure hunter is organizing their trip and wants to identify treasures that are valuable to them. Specifically, a treasure i is *valuable* if there is no other treasure j with $x_j \geq x_i$ and $y_j \geq y_i$. The treasure hunter wants to count the number of valuable treasures on their map.

(a) [10 Points] Give an efficient divide-and-conquer algorithm to count the number of valuable treasures. You may assume that two distinct treasures differ in at least one of the two parameters i.e., for distinct i and j, either $x_i \neq x_j$ or $y_i \neq y_j$.

(b) [5 Points] Analyze the running time of your algorithm (and provide the recurrence relation).

(c) [5 Points] Briefly justify why the algorithm given in part (a) is correct.

Solution to Q2

(a) Reformulation: We are given as input a set S of n distinct points $(x_1, y_1), \ldots, (x_n, y_n)$ in the plane. We say that a point (x_i, y_i) is a *maxima* point if there is no other index $j \neq i$ with $x_j \geq x_i$ and $y_j \geq y_i$. Our task is to compute the set of maxima points of S. We say that a point $p = (p_1, p_2)$ is dominated by a point $q = (q_1, q_2)$ if both $p_1 \leq q_1$ and $p_2 \leq q_2$. In other words, a point in S is a maxima point of S if it is dominated by no other point of S.
We first sort the given points in a lexicographical order i.e., in a non-decreasing order of their x-coordinate, breaking ties using the y-coordinate. Then, we execute the following algorithm:

Algorithm MaximaSet(S):

Input: A set, S, of n points in the plane
Output: The set, M, of maxima points in S

if $n \leq 1$ **then**
    return S
Let p be the median point in S, by lexicographic (x, y)-coordinates
Let L be the set of points lexicographically less than p in S
Let G be the set of points lexicographically greater than or equal to p in S
$M_1 \leftarrow$ MaximaSet(L)
$M_2 \leftarrow$ MaximaSet(G)
Let q be the lexicographically smallest point in M_2
for each point, r, in M_1 **do**
    if $x(r) \leq x(q)$ **and** $y(r) \leq y(q)$ **then**
        Remove r from M_1
return $M_1 \cup M_2$

Figure 1: Algorithm to find the Maxima Set

(c) We prove the correctness before analyzing runtime. The base case $n = 1$ is obvious. For larger n, note that any point in the set of maxima points M_2 of G is already a maxima point of S. However, a maxima point of L need not be a maxima point of S overall. Nevertheless, we observe that:

Claim. Let q be lexicographically the smallest point in M_2. If a point r in M_1 is dominated by any point in M_2, then it is dominated by q as well.

Proof. Note that since q is in M_2 (and hence in G) and r is in M_1 (and hence in L), we already know that q is lexicographically larger than r (in particular, $x(q) \geq x(r)$). So, it suffices to show

that $y(q) \geq y(r)$. Now, suppose $s = (x(s), y(s)) \in M_2$ is a point that dominates $r \in M_1$ i.e., $x(s) \geq x(r)$ and $y(s) \geq y(r)$. We now show that $y(q) \geq y(s)$, which proves the claim. Suppose not i.e., suppose that $y(q) < y(s)$. But then, as it is given that q is lexicographically the smallest point in M_2, we know $x(q) \leq x(s)$, which would imply that q is dominated by s which is a contradiction because $q \in M_2$ is a maxima point of G. □

Therefore, in order to remove the points in M_1 that are not maxima points overall, we only need to check if they are dominated by q, which is exactly what we do in the algorithm.

(b) We obtain the recurrence $T(n) \leq 2T(n/2) + O(n)$, because the "merge" step takes linear time: for every point in M_1 (whose size is at most n), we take constant time to check if it is dominated by q. Moreover, locating the median point p of S also takes linear time, as we already have a (lexicographically) sorted list of points before implementing the algorithm. Since sorting lexicographically also takes $O(n \log n)$ time, the overall time is $O(n \log n)$.

Q3 [20 Points] Two Classrooms Only!

Here is a variant on the problem of Interval Scheduling. Suppose we now have *two* classrooms available, and we want to schedule as many lectures as we can. As before, the input is a collection of intervals $[s_1, f_1), [s_2, f_2), \ldots, [s_n, f_n)$ where $n \geq 1$ and all $s_i < f_i$ are non-negative integers.
A schedule is now defined as a pair of sets (A_1, A_2), with A_1 and A_2 being the sets of lectures scheduled in classrooms 1 and 2, respectively. A schedule is said to be *valid* if it satisfies the obvious constraints:

- $A_1, A_2 \subseteq \{1, 2, \ldots, n\}$,
- $A_1 \cap A_2 = \emptyset$ (i.e., a lecture can only be scheduled in *one* classroom), and
- for all $i \neq j$ such that $i, j \in A_1$ or $i, j \in A_2$, lectures i and j do not overlap (i.e., the lectures scheduled in a classroom are compatible with each other).

(a) [8 Points] Design a greedy algorithm to solve this problem i.e., on input $[s_1, f_1), \ldots, [s_n, f_n)$, your algorithm produces a valid schedule with the maximum number of lectures.

(b) [2 Points] Analyze the running time of your algorithm.

(c) [10 Points] Prove the correctness of your algorithm.

Solution to Q3

(a) A natural greedy algorithm for this variation is as follows. First, sort the jobs in order of non-decreasing finish times. Then, handle the jobs one at a time, scheduling each job if possible. If a job can be scheduled on either processor, pick the processor where it will cause the smallest "gap". This algorithm is described in detail below, where variables E_1 and E_2 are used to keep track of the largest finish time of jobs in A_1 and A_2, respectively.

(b) The execution of the for loop takes only linear time. Therefore, the overall runtime is dominated by the sorting time, and is hence $O(n \log n)$.

Algorithm 2 Algorithm for interval scheduling in two classrooms

```
sort jobs so that f_1 ≤ f_2 ≤ ··· ≤ f_n
A_1 ← ∅
A_2 ← ∅
E_1 ← 0
E_2 ← 0
for i ← 1 to n do
    if E_2 ≤ E_1 ≤ s_i or E_1 ≤ s < E_2 then
        A_1 ← A_1 ∪ {s_i}
        E_1 ← f_i
    else
        if E_2 ≤ s_i then
            A_2 ← A_2 ∪ {s_i}
            E_2 ← f_i
        end if
    end if
end for
```

(c) Let us denote the schedule created by the algorithm after examining the first i jobs by (A_1^i, A_2^i); let the values of E_1 and E_2 be E_1^i and E_2^i. We will prove by induction on i, $0 \le i \le n$, that there is an optimal schedule (OPT_1, OPT_2) such that $A_1^i = OPT_1 \cap \{1, 2, \ldots, i\}$ and $A_2^i = OPT_2 \cap \{1, 2, \ldots, i\}$. (This implies that the algorithm produces an optimal schedule.)

Base Case: $i = 0$. This is trivially true, since any optimal schedule will work.

Induction Step: Let $0 \le i < n$, and let (OPT_1, OPT_2) be an optimal schedule such that $A_1^i = OPT_1 \cap \{1, 2, \ldots, i\}$ and $A_2^i = OPT_2 \cap \{1, 2, \ldots, i\}$.

CASE 1: The algorithm does not add job $i+1$ to either A_1^i or A_2^i.
Then job $i+1$ overlaps with a member of A_1^i and with a member of A_2^i, so $i+1$ cannot be in either OPT_1 or OPT_2. So $A_1^{i+1} = OPT_1 \cap \{1, 2, \ldots, i+1\}$ and $A_2^{i+1} = OPT_2 \cap \{1, 2, \ldots, i+1\}$.

CASE 2: $A_1^{i+1} = A_1^i \cup \{i+1\}$.
If $i+1 \in OPT_1$ we are done, so assume $i+1 \notin OPT_1$. We have $E_1^i \le s_{i+1}$.

Subcase 2A: $i+1 \in OPT_2$. (This is the most interesting case.)
The algorithm could have put job $i+1$ on processor 2 (since OPT did), and since it didn't, we must have $E_2^i \le E_1^i \le s_{i+1}$.
We can write $OPT_1 = A_1^i \cup B$ and $OPT_2 = A_2^i \cup C$ where $B, C \subseteq \{i+1, \ldots, n\}$ and $i+1 \in C$. Since the jobs were sorted according to finish time and $i+1 \in C$, every job in B starts at time $\ge E_1^i$ and every job in C starts at time $\ge s_{i+1}$. So every job in B starts at time $\ge E_2^i$ and every job in C starts at time $\ge E_1^i$. This means that we can interchange B and C, so that $(A_1^i \cup C, A_2^i \cup B)$ is a schedule; since it has the same size as $(A_1^i \cup B, A_2^i \cup C)$ it is an optimal schedule. We also have $A_1^{i+1} = (A_1^i \cup C) \cap \{1, 2, \ldots, i+1\}$ and $A_2^{i+1} = (A_2^i \cup B) \cap \{1, 2, \ldots, i+1\}$.

Subcase 2B: $i+1 \notin OPT_1$ and $i+1 \notin OPT_2$.
As in the one processor proof, since the jobs are sorted according to nondecreasing finish times, there is only one job $j \in OPT_1$ that overlaps job $i+1$, and $j > i+1$. Let $OPT_1' = OPT_1 \cup \{i+1\} - \{j\}$. Then (OPT_1', OPT_2) is an optimal schedule with the desired properties.

CASE 3: $A_2^{i+1} = A_2^i \cup \{i+1\}$. This is similar to Case 2.

Q4 [20 Points] Help Your Professor!

Your Professor has agreed to assign as much of his TAs' time as necessary to ensure that his course's Piazza page is monitored around the clock throughout the term. Suppose that his TAs have given him their availabilities, specified as intervals $[s_i, f_i)$ where $s_i < f_i$ are the start and end times of each available time slot, over some suitable range of non-negative integers.
Your Professor would like to draw up a monitoring schedule with the following properties:

- at every point in time – starting from the *earliest* TA availability to the *latest* – there is at least one TA available to monitor the course Piazza page, i.e., there are no gaps in the schedule, although it's okay if there is overlap, and
- the TAs are not overworked, i.e., the schedule uses as few available time slots as possible to achieve the first property, so that there are enough TA hours left over to help out with marking.

Note that this problem is a "dual" to the interval scheduling problem: with the same kind of input – intervals $[s_1, f_1), \ldots, [s_n, f_n)$ – we are asked a complementary question – to find a subset of intervals that totally cover the range from the earliest start time to the latest finish time, with overlap allowed, using as few intervals as possible.

(a) [5 Points] Give an ordering of the intervals for which the natural greedy algorithm corresponding to this ordering does not always find an optimal solution – include a description of some specific input for your algorithm, show the solution found by the greedy algorithm, and justify that it is not optimal.

(b) [5 Points] Design a greedy algorithm that solves this problem, i.e., on input $[s_1, f_1), ..., [s_n, f_n)$, your algorithm either produces a valid schedule that uses the smallest number of intervals, or it correctly states that no such schedule is possible.

(c) [2 Points] Analyze the running time of your algorithm.

(d) [8 Points] Prove the correctness of your algorithm.

Solution to Q4

(a) Consider the intervals from longest to shortest, i.e., sort them so that $f_1 - s_1 \geq f_2 - s_2 \geq \cdots \geq f_n - s_n$, and use the natural greedy algorithm: for each interval, in order, put the interval in the solution if it covers any part of the schedule not already covered.

Counter-example: Let the input intervals be $I_1 = [1, 5), I_2 = [4, 10), I_3 = [2, 9)$. Greedy by length will pick I_3 first, then both I_1 and I_2 (to cover the start and end periods of the schedule). However, it was possible to cover the entire schedule using only I_1 and I_2.

(b) We describe the algorithm (Algorithm 3) below.

Algorithm 3 Algorithm for interval scheduling dual

Require: $I_1 = [s_1, f_1), \ldots, I_n = [s_n, f_n)$
Ensure: A valid schedule with maximum number of lectures

```
1: sort the intervals by finish time (f_1 ≤ ··· f_n)
2: S ← ∅ # intervals selected so far
3: E ← min(s_1, ..., s_n) # earliest start time of intervals in S
4: B ← 1 # index of first interval still under consideration
5: while B ≤ n and E < f_n do
6:     # find the interval with the latest finish time that
7:     # overlaps with the intervals in S
8:     i ← n
9:     while i ≥ B and s_i > E do
10:        i ← i − 1
11:    end while
12:    if i < B then
13:        return "not possible" # unavoidable gap in the schedule
14:    end if
15:    # update S, E, and B: remove from consideration all intervals whose finish time is no later
       than E
16:    S ← S ∪ I_i
17:    E ← f_i
18:    B ← i + 1
19:    while f_B ≤ E do
20:        B ← B + 1
21:    end while
22: end while
```

(c) Runtime: $O(n \log n)$ to sort in line 1. The main loop iterates at most n times (at each iteration, B increases by at least 1) and each iteration requires worst-case time $O(n)$ (line 6), for a total of $O(n^2)$.

(d) Correctness: Suppose $I_1, \ldots, I_n$ is an input for which there is no solution. This means that even including every interval, there is at least one "gap" in the schedule. Let $[t_1, t_2)$ be the earliest such gap, i.e., there are intervals that completely cover $[\min(s_1, \ldots, s_n), t_1)$ but no interval that overlaps with $[t_1, t_2)$. Then, the greedy algorithm will eventually include some interval whose finish time is t_1, and it will be unable to find any interval with a later finish time that overlaps, causing it to return "not possible" on line 13.
Now, suppose that there is a solution, and let $S_0, \ldots, S_n$ be the sequence of partial solutions constructed by the greedy algorithm. We prove that for each $i \in [0, \ldots, n]$, S_i is promising – i.e., there is some optimal solution O_i that extends S_i (in the sense that O_i contains each interval from

S_i and any additional intervals in O_i belong to the set $\{I_{B_i}, \ldots, I_n\}$).

Base Case: $S_0 = \emptyset$ is promising because every optimal solution contains the intervals in S_0 together with other intervals from $\{I_1, \ldots, I_n\}$.

Induction Hypothesis: Suppose $i \in [0, \ldots, n]$ and S_i is promising, i.e., optimal solution O_i extends S_i.

Induction Step: Let $S_{i+1} = S_i \cup \{I_j\}$. If $I_j \in O_i$, then O_i extends S_{i+1}. Otherwise, let I_k be an interval in O_i that overlaps with the last interval in S_i, i.e., such that $s_k \leq E_i$. Because of the order in which intervals are considered by the greedy algorithm, it must be the case that $k < j$, i.e., $f_k \leq f_j$. Then, $O_{i+1} = O_i \cup \{I_j\} \setminus \{I_k\}$ is an optimal solution that extends S_{i+1}: it contains no more intervals than O_i, and still covers the entire schedule. (Intuitively, if there is an optimal solution that uses any interval other than the greedy one to cover a certain portion of the schedule, that interval can be replaced with the greedy one without danger of creating gaps.)
Hence, the solution returned by the greedy algorithm must be optimal, since it is promising, but with no intervals left over for possible extension at the end.

Bonus Question

Q5 [20 Points] Voltage Optimization!

You have recently branched out into manufacturing bulbs! Your R&D department has produced several different designs. They don't know how much voltage each design can withstand, but using some physics, they can compare different designs in terms of their maximum voltage.

You have, in front of you, n bulbs sorted in a *non-increasing* order of the maximum voltage they can withstand. Your village uses 120V. You want to find the index r such that bulbs 1 through r can withstand 120V, but bulbs $r+1$ through n cannot. In $O(1)$ time, you can test any bulb at 120V. Your first thought is to do a binary search in $O(\log n)$ time, but the problem is that every time you test a bulb that *cannot* withstand 120V, the bulb burns out. You want to limit the number of bulbs wasted. Of course, you can do a linear search and solve the problem while wasting at most one bulb, but that is too slow.

Suppose you are willing to waste at most k bulbs, where k is a constant. Design an algorithm for this problem. What is the asymptotic number of bulbs your algorithm tests in the worst case as a function of k?

[Note: As a warm start, design an $O(\sqrt{n})$ algorithm for $k = 2$. To receive any partial credit, you must solve the problem for at least $k = 3$ with an $o(\sqrt{n})$ algorithm.]

Solution to Q5

Let $D[1 \ldots n]$ denote the array where $D[i]$ is the maximum voltage that bulb i can withstand. Hence, we can make the query $D[i] \stackrel{?}{\geq} 120V$ in $O(1)$ time.

Notes to students: For easier understanding, I have written the algorithm below in the "bad" manner, which takes an entire array as input and calls itself recursively on explicit subarrays. Hence, the algorithm returns a "bulb" rather than its "index". You can convert this into a recursive algorithm that accesses D as a global array, only takes a start and an end index as input, and searches in that range within the global D array. This implementation would not only be truly efficient, but also be able to return the index of the bulb found. Also, I will assume that at least bulb 1 can withstand 120V, but the algorithm can be easily modified to detect if this is not the case.

We show, via induction on $k \geq 1$, that Algorithm VOLTAGE$_k$ returns the bulb with the greatest index which can withstand 120V, burns at most k bulbs, and makes $O(n^{\frac{1}{k+1}})$ queries.

Algorithm 4 VOLTAGE$_k(D[1 \ldots n])$

```
1:  if k = 0 then
2:      for i = 1, ..., n do
3:          If D[i] < 120V (bulb burns out), return the bulb at index i − 1
4:          Return the bulb at index n
5:      end for
6:  else
7:      for t = 1, ..., n^{1/(k+1)} − 1 do
8:          i ← t · n^{k/(k+1)} + 1
9:          If D[i] < 120V (bulb burns out), call VOLTAGE_{k−1}(D[(t − 1) · n^{k/(k+1)} + 1 ... t · n^{k/(k+1)}])
10:     end for
11:     Call VOLTAGE_{k−1}(D[n − n^{k/(k+1)} + 1 ... n])
12: end if
```

In the base case of $k = 1$, the algorithm simply does a linear search in the decreasing order of the maximum voltage. Hence, it will find the promised bulb after at most one bulb burning out. This takes $O(n)$ queries.

Suppose VOLTAGE$_{k-1}$ finds the correct bulb with at most at most $k-1$ bulb burning out and in $O(n^{\frac{1}{k}})$ queries given n bulbs.

In VOLTAGE$_k$, it is easy to see that Line 9 makes $O(n^{\frac{1}{k+1}})$ queries. Further, since the bulbs are sorted in the decreasing order of their maximum voltage, either some bulb will burn out in that line at some index t, in which case we know that desired bulb is in $D[(t-1) \cdot n^{\frac{k}{k+1}} + 1 \ \ldots \ t \cdot n^{\frac{k}{k+1}}]$ since the bulb corresponding to index $t-1$ did not burn out (and the algorithm correctly calls itself recursively on this range), or none of the bulbs burn out, in which case we know that the desired bulb is in the final range $D[n - n^{\frac{k}{k+1}} + 1 \ \ldots \ n]$, in which case the algorithm correctly calls itself on this range. By our induction hypothesis, the recursive call will make $O(n^{\frac{k}{k+1} \cdot \frac{1}{k}}) = O(n^{\frac{1}{k+1}})$ queries. Adding the queries made in Line 9, we have that the total number of queries is also $O(n^{\frac{1}{k+1}})$. Further, the algorithm burns at most one bulb before recursively calling VOLTAGE$_{k-1}$, which burns at most $k-1$ bulbs. Hence, the overall algorithm burns at most k bulbs.