CSCI 3104
Spring 2018
Problem Set 6

Allison, George
02/17

March 1, 2018

1. a) Algorithm A will show more ads than minimum in any situation where there are multiple overlapping groups of simultaneously-present wizards and the 'inner' groups have more wizards than the 'outer' groups. For an example of this configuration see **Figure 1** (at the end of this document).

b) S1) $t_1 \leq e_j$ can be proven by showing that if t_1 was greater than e_j wizard W_j would have left before seeing an ad, since t_1 is the first advertisment.

S2) If t_1 was to be deleted, while $t_1 < s_j$ and it resulted in a wizard not seeing an ad, that would mean said wizard (which we can refer to as W_x) would need to have left before W_j left. Unless $s_x < e_j$ the deletion of t_1 would make no changes, and $s_x < e_j$ goes against the given rule that $s_j \leq$ all other departure times.

S3) In the case where $t_1 = e_j$ a vaild solution would still remain unless that modification caused some wizard (again, referred to as W_x) to be uncovered. For W_x to be uncovered it would need to have a departure time before the start time of W_j. Similarly to (S2) this goes against the given rule that $s_j \leq$ all other departure times.

c) i) An optimal greedy algorithm would first check if all the wizards can be taken care of with one ad. If not, it would divide the n wizards in half and recursively check each subdivison until the optimal amount of ads are found.
The comparisons between wizards can be made by checking for duplicate numbers between time slots, for example, Harry's availability of [6, 60] and Ron's availibility of [6, 99] would have duplicates of [6, 7, 8, ... 59, 60]. By comparing arrays of availibility, the algorithm would eventually determine a duplicate, or would recursively check smaller subdivisions until the base case of a single wizard is reached. Here is the psuedocode for this algorithm:

```
//Java program for scheduling the optimal number of ads for the wizards

// Assuming that we are using this sort of Tuple data type for handling time intervals:
private static class Tuple<s,l>{
   public s; // time of arrival
   public l; // time of departure
}
```

```
public static boolean compare(Tuple a, Tuple b){
   //create an array of duplicates and save the duplicates during each comparison of each
       subdivision
}

public static int[] adScheduler(Tuple[] timeInts){
   int i = 0;
   while(i < timeInts.length()){
      if(!compare(timeInts[0], timeInts[i]))
         break;
      if(i == timeInts.length())
         return /* in this case one ad will suffice, all wizards overlap */
   }
   adScheduler(Tuple[/* from index 0 to length / 2 */]); //recursive call
   adScheduler(Tuple[/* from index length / 2 to length */]); //recursive call
}
```

---

ii) The algorithm will surely provide the optimal solution because it checks each possiility of ad placement, starting at one ad and checking every possibility. The running time of the algorithm is $\Omega(n)$ and $2^{O}(n)$.

iii) Applied to n = 6 example:

-When comparing Ron and Hermione in the first pass, there will be no duplicates so the group will be spilt in half and recursively run again.

-In the second pass the first half of the wizards (Minerva, Harry, and Ron) will all share duplicates, one of which will be chosen for an ad placement. The second half of the wizards will also share duplicates, so the algorithm would successfully result in the optimal number of 2 advertisments.

2. a) The algorithm below computes the minimum-size subset in $O(nlog(n))$ time by utilizing quicksort to sort the time ranges by their arrival time s. After, it operates on the set by comparing arrival and departure times to choose coverage; this results in n running time leaving the total asymptotic time complexity as $O(nlog(n))$. The following psuedocode outlines how the problem can be solved.

```
/*Step 1) Sort S[] with Quicksort (O(nlog(n))) to be in ascending order by start times
    (s_i)
Step 2) Iterate through the sorted list checking element [n] with its predecessor [n+1],
    checking for coverage:
Psuedocode for a Java program:*/

// Assuming that we are using this sort of Tuple data type for handling coverage times:
private static class Tuple<s,l>{
   public s; // time of arrival
   public l; // time of departure
}

//function that takes array S of Tuple time ranges and handles the scheduling:
public static Tuple[] myScheduler(Tuple[] S){
   myQuicksort(S); //quicksort S in order of arrival time from lowest to highest
   Tuple[] T = new Tuple[]; //array that we will return
   T[0] = S[0]; //add earliest element to T[]

   int pivot = 0;

   for(int i = 1; i < S.length(); i++){
      if(S[i].l > T[pivot].l){ //if the element's departure is later than pivot
         if(S[i].s <= T[pivot].l){ //if the element's start time is earlier than pivot
           if(S[i].l > T[pivot+1].l) //compare departure time with T[pivot + 1]
            T[pivot+1] = S[i]; //swap if longer duration
         }
         if(S[i].s > T[pivot].l){ //if element's start time is after pivot's departure
            pivot++;
            T[pivot] = S[i]; //update pivot point to the new element
         }
      }
   }
}
```

b) The above algorithm can ensure a minimum-size covering subset due to the comparisons it makes to the elements once they are sorted. With the elements sorted in ascending order by their start time (s) the algorithm assigns pivot points to determine the longest possible continuation of coverage. Every element that overlaps with the pivot element is compared and only the best one is chosen to be added to the coverage array T. Once all options are exhausted, the newest element (or in the case of a coverage gap, the next element) becomes the pivot point and the process repeats. The strategy of using pivot points is convenient because it handles both possible scenarios of encountering inefficient elements or gaps in coverage, and it only needs to pass through the list of elements one time (albeit, while making multiple comparisons to each element) to achieve the optimal solution.

3. a) A recurrence relation for the following program is $T(n) * T(n) + n$ resulting in $O(n^2)$

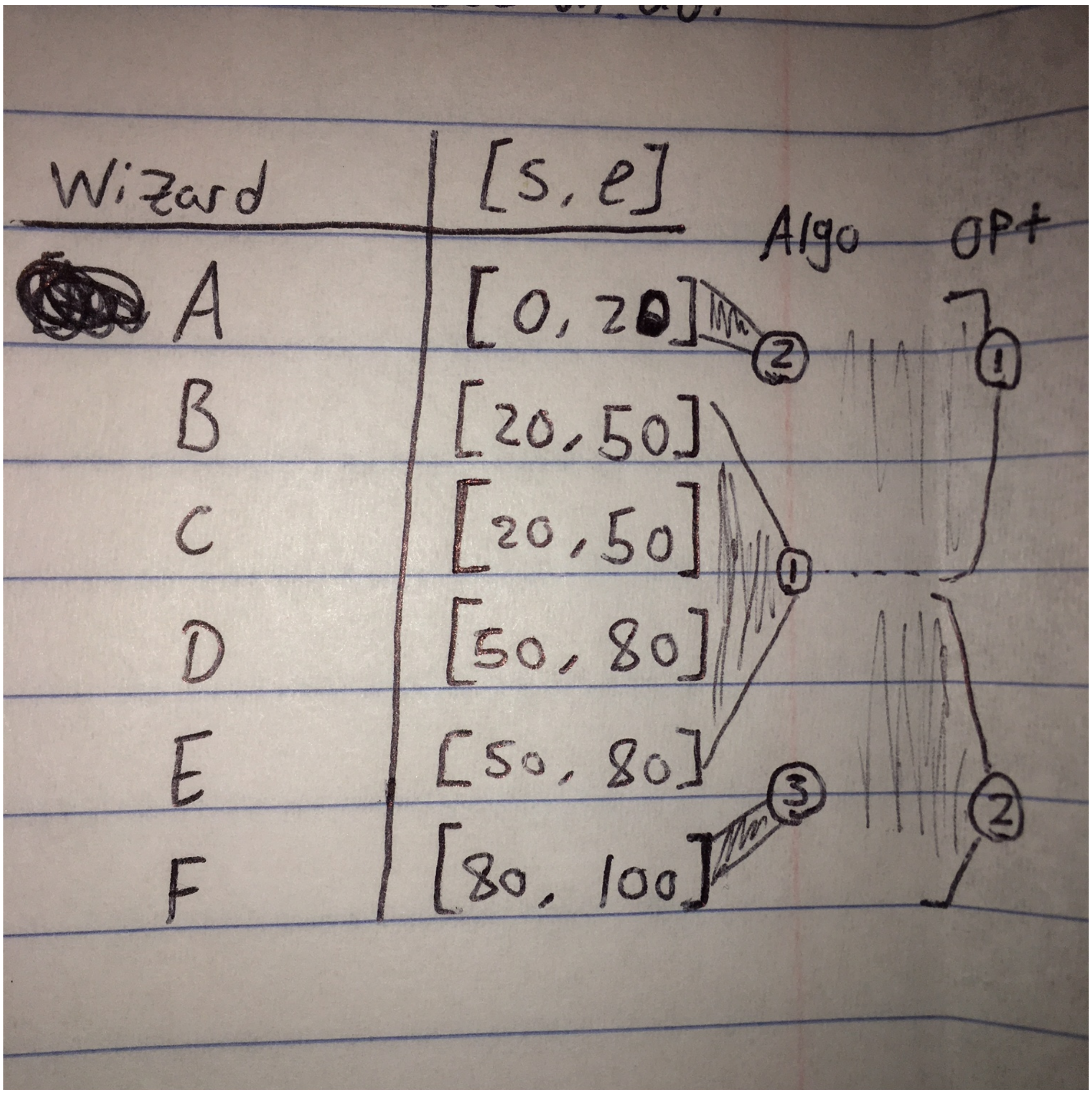
```
C(i, n, b){
   int x;
   while(d[i] > n){ i--; }
   if(b ==0 || i <= 2){
      while(i > 0){
         x = n / d[i];
         n = n % d[i];
         i--;
      }
      return x;
   } else {
      int na = i - 1;
      while(i > 0){
         if(i != na){
            x = n / d[i];
            n = n % d[i];
```

```
        i--;
      }
    }
    return x;
  }
}
```

---

b) I don't know.

c) I don't know.



Wizard	[s, e]
A	[0, 20]
B	[20, 50]
C	[20, 50]
D	[50, 80]
E	[50, 80]
F	[80, 100]

Algo
Opt
2
1
3
1
2

Figure 1: This image displays an assortment of wizards for problem 1a. When the greedy algorithm is run on this set of wizards it results in ads being shown 3 times, shown by the column labeled "Algo". The optimal number of ads shown is only 2, shown under "opt". Any assortment of wizards where the groups overlap and decrease in number outward will result in a superoptimal number of ads shown.