Problem 2

Consider a variant on the problem of Interval Scheduling where instead of wanting to schedule as many jobs as we can on one processor, we now want to schedule ALL of the jobs on as few processors as possible.
The input to the problem is $(s_1, f_1), (s_2, f_2), , (s_n, f_n)$, where $n \geq 1$ and all $s_i < f_i$ are non-negative integers. The integers s_i and f_i represent the start and finish times, respectively, of job i.
A schedule specifies for each job i a positive integer $P(i)$ (the processor number for job i). It must be the case that if $i \neq j$ and $P(i) = P(j)$, then jobs i and j do not overlap. We wish to find a schedule that uses as few processors as possible, i.e., such that $max\{P(1), P(2), \cdots, P(n)\}$ is minimal.

1. Design an algorithm to solve the problem in time $O(n^2)$, i.e. strictly less than $O(n^2)$

   (a) Let P be an empty array of size n , representing $P(i)$ at index i
   (b) Sort (s_i, f_i) by start time such that for all $i \leq j$, $s_i \leq s_j$
   (c) Starting from the start of the sorted job array J and for each job j_i do
      i. Starting from processor number $k = 1$
      ii. Define $J_k \subseteq J$ such that for all $j \in J_k$, $P(j) = k$. If j_i is compatible with all $j \in J_k$, then assign j_i processor number k, i.e. let $P(i) \leftarrow k$
      iii. Otherwise, increment k and try the previous step again until j_i is assigned to either a previously used processor number or a new processor number not used before.
   (d) Return P

2. Prove that the above algorithm is guaranteed to compute a schedule that uses the minimum number of processors.

   We will prove that the greedy choice is always in some optimal solution to the problem. Then we prove that the problem exhibits optimal substructure. Here we define a *compatible* processor number k for job j be an integer such that all jobs previously assigned k are compatible with j. Let J be the input jobs given. Let $J_t := \{j_i \in J : s_i \geq s_t\}$ be subset of J such that all jobs in J_t starts after j_t starts. Let $Max(P)$ be the maximum of processor numbers in P

   Proposition. *Consider any subproblem J_t, let $j_i \in J_t$ be the job with earliest starting time, and let be k be the lowest compatible processor number with j_i. Then assigning k to j_i is in some optimal ($max\{P(1), \cdots, P(n))\}$ minimized) solution to J_t*

Proof. Assume P' is an arbitrary optimal solution to J_t. Let $k' = P'(i)$. If $k = k'$, then we are done the proof since k is assigned to j_i by the greedy choice, which is in the optimal solution P'. Otherwise if $k \neq k'$, since k is the lowest processor number possible (i.e. $k \leq k'$), then $k < k'$. Now we can construct a solution $P = P'$ where $P(i) = k$, thus $P(i) < P'(i)$. We arrive at a contradiction on the assumption that P' is optimal. Hence we conclude that the greedy choice is always in some optimal solution to J_t □

Proposition. *The scheduling problem exhibits optimal substructure.*

Proof. Given arbitrary index i, we separate the problem into a greedy choice and a single subproblem, i.e. $\{j_i\}$ and $J_{after} = J_i$. We make the choice assigning a processor number k to j_i, Assume such assignment is in some optimal solution to the problem P'. Now we are left with assigning processor number to J_{after} with P_{after}. Then the optimal solution follows

$$Max(P') = Max\{k, Max(P_{after})\}$$

We claim that if P' is optimal, then P_{after} is also optimal, in a sense that if $Max(P_{after}) > k$, then $Max(P_{after})$ is minimized. If $Max(P_{after}) \leq k$, then solution is already optimal. Otherwise if $Max(P_{after}) > k$, then suppose we can find a more optimal solution P''_{after} such that $Max(P''_{after}) < Max(P_{after})$ then we can substitute P''_{after} for P_{after} and construct another solution set P'' with

$$Max(P'') = Max\{k, Max(P''_{after})\} < Max\{k, Max(P_{after})\} = Max(P')$$

Hence contradicting the optimality assumption for P', hence P_{after} must be optimal in itself. □

We conclude by combining propositions proved earlier. By optimal substructure of the problem, given that at each step the greedy choice is optimum and we are left with finding optimal solution to a smaller subproblem, i.e. J_{after}, the solution to the original solution is optimal, specifically, the algorithm uses minimum number of processors.

3. Briefly describe an efficient implementation of the algorithm, making it clear what data structures you are using. Express the running time of your implementation as a function of n (the number of jobs), using appropriate asymptotic notation.

We will use a min heap H to store an array of finish time of currently scheduled jobs.

$Q.size$ is size of the heap and assume is updated during insertion and deletion.

```
1  Function Schedule-All (s, f)
       Input: s, f are arrays of size n, representing job j_i = (s_i, f_i) at index i
       Output: P is an array of size n storing P(i) at index i, where
               max{P(1), ··· , P(n)} is minimized
2      P ← Array of size n
3      H ← Min-Heap
4      Sort s, f by start time together such that s_i ≤ s_j for all i ≤ j
5      for i = 1 to n do
6          while Heap-Maximum (H) < s_i do
7              Heap-Extract-Max (H)
8          Heap-Insert (H, f_i)
9          P(i) ← H.size
10     return P
```

At line 6-7, we remove jobs' finish time from the heap H such that the heap retains previously scheduled jobs that overlaps j_i. The size of the heap represent the smallest compatible processor number, which we record in solution $P(i)$ at each iteration afer inserting the finish time of j_i to the heap.

Now we analyze running time

(a) Sorting takes $O(n \lg n)$

(b) By the time the procedure terminates, each jobs' finish time is inserted and removed from the heap, since HEAP-EXTRACT-MAX and HEAP-INSERT has worst case running time of $O(\lg n)$. Heap insertion and deletion has worst case running time of $O(2n \lg n) = O(n \lg n)$

(c) HEAP-MAXIMUM is called at least once per iteration of for loop for a total of n iterations; and it is called at most n number of times for each successful condition evaluation and subsequent deletion operation (since at most deleting a total of n items). HEAP-MAXIMUM has worst case running time of $O(1)$ hence by the time procedure terminates, heap lookup operation has a worst case running time of $O(2n) = O(n)$

(d) Assigning P at index i takes $O(1)$ each iteration and since there are n iterations, has a worst case running time of $O(n)$

(e) To conclude, the algorithm has a worst case running time of $O(n \lg n)$

Problem 3

Here is another variant on the problem of Interval Scheduling. Suppose we now have two processors, and we want to schedule as many jobs as we can. As before, the input is

$(s_1, f_1), (s_2, f_2), , (s_n, f_n)$, where $n \geq 1$ and all $s_i < f_i$ are nonnegative integers. A *schedule* is now defined as a pair of sets (A_1, A_2), the intuition being that A_i is the set of jobs scheduled on processor i. A schedule must satisfy the obvious constraints: $A_1 \subseteq \{1, 2, \cdots, n\}$, $A_2 \subseteq \{1, 2, \cdots, n\}$, $A_1 \cap A_2 = \emptyset$, and for all $i \neq j$ such that $i, j \in A_i$ or $i, j \in A_2$, jobs i and j do not overlap.

1. Design an algorithm (write a pseudocode) to solve the above problem in time $O(n^2)$, i.e., strictly less than $O(n^2)$.

   Let $J : \{1, 2, \cdots, n\}$ be the input set of jobs given. Here we define that a job $j \in J$, having start time s, is *compatible* with A_i or processor i if

   $$\forall j \in A_i : f_j \leq s$$

   Let subproblem J_t be a set of jobs such that

   $$\forall j \in J_t : f_t \leq s_j$$

   in otherwords, J_t is the set of jobs that starts after job t ends.
   We define waste time W_i for a job j, having start time s, with respect to a compatible A_i as

   $$W_i = s - \underset{j \in A_i}{Max}\{f_j\}$$

   in other words, the waste time is the time period between the finish time of the last finishing jobs already in A_i and the start time of the job j in consideration

   (a) Let (A_1, A_2) be set of jobs scheduled on processor 1 and 2 respectively.
   (b) Sort (s_i, f_i) by finish time such that for all $i \leq j$, $f_i \leq f_j$
   (c) Starting from the start of the sorted job array and for each job $j \in J$ do
      i. Test if j is compatible with A_1 and A_2.
      ii. If j is not compatible with either processor set, continue to next iteration
      iii. If j is compatible with only one of A_1 and A_2, then add j to the set of jobs scheduled on the compatible processor
      iv. If j is compatible with both A_1 and A_2, then add j to the set of jobs A_i such that waste time for job j with respect to A_i, W_i, is minimized
   (d) Return (A_1, A_2)

2. Prove that the above algorithm is guaranteed to compute an optimal schedule.

   Proposition. *Consider any subproblem J_t, let $j_e \in J_t$ be the job with earliest finish time with waste time W_1 and W_2. Then making the choice of assigning j_e described above is in some optimal (i.e. $|A_1| + |A_2|$ maximized) solution to the problem.*

Proof. Let (A_1, A_2) be some optimal solution to the subproblem J_t. Let $j_e \in J_t$ be job with earliest finish time. By definition of J_t, j_e is compatible with at least one of A_i. Suppose j_e is compatible with exactly one of A_i, the claim is true based on proof of correctness of interval scheduling on a single processor in class. Otherwise, j_e is compatible with both A_1 and A_2. Without loss of generality, suppose $W_1 < W_2$, hence the greedy choice assigns j_e to A_1. Let j_m be the job in A_1 with earliest finish time and similarly let j_n be such for A_2. Now consider,

(a) If $j_e = j_m$, optimal solution contains the greedy choice, claim true

(b) If $j_e \neq j_n$, then

   i. $j_e \in A_2$, since j_n is earliest finishing job in A_2, $j_e = j_n$. We can construct a new solution set (A'_1, A'_2) such that

□

3. Briefly describe an efficient implementation of the algorithm, making it clear what data structures you are using. Express the running time of your implementation as a function of n (the number of jobs), using appropriate asymptotic notation.