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 $max\{P(1), \cdots, P(n)\}$ is minimized
2   $P \leftarrow$ Array of size n
3   $H \leftarrow$ Min-Heap
4   Sort s, f by start time together such that $s_i \leq s_j$ for all $i \leq j$
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     $P(i) \leftarrow H.size$
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 $J_s \subseteq J$ if j starts after every job in J_s finishes, in other words,

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

   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 set of jobs J_s as

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

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

   (a) Let $(A_1, A_2) = (\varnothing, \varnothing)$
   (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 set, continue to next iteration
      iii. If j is compatible with only one of A_1 and A_2, then add j to the compatible set
      iv. If j is compatible with both A_1 and A_2, then add j to 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 (O_1, O_2) be some optimal solution to the original problem J. Let $(A_1 \subseteq O_1, A_2 \subseteq O_2)$ be the optimal solution to the subproblem J_t with respect to (O_1, O_2). and let $(B_1 \subseteq O_1, B_2 \subseteq O_2)$ be the optimal solution to subproblem $J \setminus J_t$. 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 B_i.

(a) Suppose j_e is compatible with exactly one of B_i, without loss of generality, suppose B_1 is the compatible set and B_2 is the in-compatible set. Let $j_a \in A_1$ be the first finishing job in A_1. Then we have,

   i. If $j_e = j_a$, then the proposition holds

   ii. If $j_e \neq j_a$. Since j_e is not compatible with B_2, $j_e \notin A_2$. Then consider a new solution set $A'_1 = A_1 \cup \{j_e\} \setminus \{j_a\}$. Note A_2 is unchanged. jobs in A'_1 are disjoint because A_1 is disjoint, $j_a \in A_1$ is the first job to finish and $f_{j_e} \leq f_{j_a}$. Since $|A'_1|+|A_2| = |A_1|+|A_2|$, (A'_1, A_2) is an optimal solution to subproblem J_t that contains j_e, hence the proposition holds.

   Similar argument holds if B_2 is the compatible set

(b) Now consider the case where j_e is compatible with both B_1 and B_2. Without loss of generality, suppose $W_1 < W_2$, hence the greedy algorithm assigns j_e to B_1. Let $j_a \in A_1$ be the earlist finishing job in A_1; let $j_b \in A_2$ be the earlist finishing job in A_2,

   i. If $j_e = j_a$, then the proposition holds

   ii. If $j_e \neq j_a$, Since j_e is compatible with B_2 as well as B_1 there are two cases as to where j_e might end up

      A. If $j_e = j_b$, then consider a new solution set where $A'_1 = A_2$ and $A'_2 = A_1$, i.e. switching the set of jobs for processor 1 and 2. Note A'_1 and A'_2 are disjoint sets of jobs since A_1 and A_2 are disjoint sets. and $|A'_1| + |A'_2| = |A_1| + |A_2|$. Since (A_1, A_2) optimal, then (A'_1, A'_2) are optimal solutions and that A'_1 now contains j_e, because $j_e = j_b \in A_2 = A'_1$. The proposition hence holds.

      B. If j_e is not in either A_1 or A_2, then consider a new solution $A'_1 = A_1 \cup \{j_e\} \setminus \{j_a\}$. Proposition holds with same argument provided in $(a).ii$.

   Similar argument holds if $W_1 > W_2$.

Proposition. *This scheduling problem exhibits optimal substructure.*

Proof. Given arbitrary index t, we separate the problem into a greedy choice and a single subproblem, i.e. $\{j_e\}$ and J_t. Let (A_1, A_2) be the optimal solution for J_t. We make the greedy choice of adding j_e to A_i or skipping j_e (i.e. $j_e = \emptyset$) to maximize time as a resource for subsequent jobs, and to minimize waste time as a resource if there is a choice to select one assuming both processors are available at the time. Assume such greedy choice is optimal, we are left with processing a smaller subproblem

$J_{after} = J_t \setminus \{j_e\}$. We claim that if (A_1, A_2) is optimal, then solution to subproblem J_{after}, (C_1, C_2) must also be optimal. Consider an alternative solution (C_1', C_2') that is even more optimal, i.e. $|C_1'| + |C_2'| \geq |C_1| + |C_2|$. We can construct a new solution by replacing (C_1, C_2) with (C_1, C_2) and get an overall more optimal solution $(A_1' = \{j_e\} \cup C_1', A_2' = \{j_e\} \cup C_2')$ such that

$$|A_1'| + |A_2'| < |A_1| + |A_2|$$

Contradicts assumption that (A_1, A_2) is optimal. Hence solution to subproblem J_{after} must be optimal □

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 schedules most jobs on two 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.

   Let A_1 and A_2 be two linked list. Job j may be appended to the tail of A_1 or A_2 in constant time $O(1)$. Assume that the last job added to A_i can be efficiently looked up in $O(1)$ time with $A_1.tail$ operation.

```
Function Compatible (j, A1, A2, s, f)
    Output: Returns None if j not compatible with A1 or A2, Return Both if j is
            compatible with both A1 and A2, and return the processor number,
            either 1 or 2, if j is compatible with only A1 or A2. Each return
            statement also include the computed waste time W1 and W2
    diff-one = s[j] − f[A1.tail]
    diff-two = s[j] − f[A2.tail]
    if diff-one > 0 and diff-two > 0 then
        return (Both, diff-one, diff-two)
    else if diff-one ≤ 0 and diff-two ≤ 0 then
        return (None, diff-one, diff-two)
    else
        if diff-one ≥ 0 then
            return (1, diff-one, diff-two)
        else
            return (2, diff-one, diff-two)
Function Schedule-On-Two-CPU (s, f)
    Input: s, f are arrays of size n, representing job j_i = (s_i, f_i) at index i
    Output: (A1, A2) is a set of solution to the problem given
    A1, A2 ← Linked-List
    A1.append(1)        // Add first finishing job arbitrarily to A1
    for j = 2 to n do
        (T, W1, W2) = Compatible(j, A1, A2, s, f)
        if T is Both then
            if W1 < W2 then
                A1.append(j)
            else
                A2.append(j)
        else if T is 1 then
            A1.append(j)
        else if T is 2 then
            A2.append(j)
    return (A1, A2)
```

Now we analyze running time. There are $O(n)$ iterations, and in each iteration, at most one *append* operation and *tail* operation, each $O(1)$, is required for Linked List operation. There is also some $O(1)$ array lookup in s and f. Hence the algorithm has a worst case running time of $O(n)$

Problem 5

During the renovations at Union Station, the work crews excavating under Front Street found veins of pure gold ore running through the rock! They cannot dig up the entire area just to extract all the gold: in addition to the disruption, it would be too expensive. Instead, they have a special drill that they can use to carve a single path into the rock and extract all the gold found on that path. Each crew member gets to use the drill once and keep the gold extracted during their use. You have the good luck of having an uncle who is part of this crew. Whats more, your uncle knows that you are studying computer science and has asked for your help, in exchange for a share of his gold!

The drill works as follows: starting from any point on the surface, the drill processes a block of rock $10cm \times 10cm \times 10cm$, then moves on to another block $10cm$ below the surface and connected with the starting block either directly or by a face, edge, or corner, and so on, moving down by $10cm$ at each step. The drill has two limitations: it has a maximum depth it can reach and an initial hardness that gets used up as it works, depending on the hardness of the rock being processed; once the drill is all used up, it is done even if it has not reached its maximum depth.

The good news is that you have lots of information to help you choose a path for drilling: a detailed geological survey showing the hardness and estimated amount of gold for each $10cm \times 10cm \times 10cm$ block of rock in the area. To simplify the notation, in this homework, you will solve a two-dimensional version of the problem defined as follows.

- **Input** A positive integer d (the initial drill hardness) and two $[m \times n]$ matrices H, G containing non-negative integers. For all $i \in \{1, \cdots, m\}$, $j \in \{1, \cdots, n\}$, $H[i, j]$ is the hardness and $G[i, j]$ is the gold content of the block of rock at location i, j (with $i = 1$ corresponding to the surface and $i = m$ corresponding to the maximum depth of the drill). There is one constraint on the values of each matrix: $H[i, j] = 0 \implies G[i, j] = 0$ (blocks with hardness 0 represent blocks that have been drilled already and contain no more gold).
- **Output** A drilling path $j_1, j_2, \cdots, j_l$ for some $l \leq m$ such that:
  1. $1 \leq j_k \leq n$ for $k = 1, 2, \cdots, l$ (horizontal coordinate is valid)
  2. $j_{k-1} - 1 \leq j_k \leq j_{k-1} + 1$ for $k = 2, \cdots, l$ (each block is underneath the one just above, either directly or diagonally, always going down)
  3. $H[1, j_1] + H[2, j_2] + \cdots + H[l, j_l] \leq d$ (the total hardness of all the blocks on the path is no more than the initial drill hardness)
  4. $G[1, j_1] + G[2, j_2] + \cdots + G[l, j_l]$ is maximum (the path collects the maximum amount of gold possible)

Solution. □

1. **optimal substructure** Let $O_n = \{j_1, \cdots, j_l\}$ be the optimal solution to the problem given. Let $OPT(n, d)$ be the maximum amount of gold to the optimal solution under

the hardness limit d, i.e.

$$OPT(n,d) := \sum_{k=1}^{l} G[k, O_n[k]] \quad \text{such that} \quad \sum_{k=1}^{l} H[k, O_n[k]] \leq d$$

For every path j possible, either $j \in O_n$ or $j \notin O_n$

(a) If $j_k \notin O_n$, then we consider a smaller subproblem with the same hardness limit d. Since the drill moves one unit down and $v = \{-1, 0, 1\}$ unit sideways, the possible path is therefore $[k-1, j_k + v]$. The optimal value is therefore given by the maximum optimal value of the subproblems

$$OPT(k, j_k, d) = \underset{v \in \{-1,0,1\}}{Max} \{OPT[k-1, j_k + v, d]\}$$

(b) If $j_k \in O_n$, then we consider a smaller subproblem with a reduced hardness limit $d - H[k, j_k]$ since we have added j_k to the optimal solution. The optimal value for j_k is therefore given by the maximum optimal value of the subproblems with reduced hardness limit in addition to the amount of gold contributed by drilling j_k

$$OPT(k, j_k, d) = \underset{w \in \{-1,0,1\}}{Max} \{G[k, j_k] + OPT[k-1, j_k + w, d - H[k, j_k]]\}$$

(c) Therefore,

$$OPT(k, j_k, d) = \underset{v,w \in \{-1,0,1\}}{Max} \{OPT[k-1, j_k + v, d], G[k, j_k] + OPT[k-1, j_k + w, d - H[k, j_k]]\}$$

2. **Define array to store computed values** Now we consider storing previously computed values in an array $M[0 \cdots m, 0 \cdots n, d]$, where $M[i, j, d]$ holds the optimal value for all path $\{j_1, \cdots, j_k\}$, where $k = i$ and $j_k = j$. in other words, the largest amount of gold under hardness restriction at $[i, j]$ via any reachable path from surface.

3. **Redefine recurrence relation in terms of array** Now we can re-define $M[i, j, d]$ recursively as follows

$$M[i,j,d] = \begin{cases} 0 & \text{if } i = 0 \\ \underset{v \in \{-1,0,1\}}{Max} \{M[i-1, j+v, d]\} & \text{if } H[i,j] > d \\ \underset{v,w \in \{-1,0,1\}}{Max} \{M[i-1, j+v, d], G[i,j] + M[i-1, j+w, d - H[i,j]]\} & \text{if } H[i,j] \leq d \end{cases}$$

4. **Bottom-Up Approach**

`Drill-Gold` (d, H, G)
$M \leftarrow [0 \cdots m, 0 \cdots n, d]$
for $j = 0$ **to** n **do**
    for $w = 0$ **to** d **do**
        $M[0, j, w] \leftarrow 0$
for $i = 1$ **to** m **do**
    for $j = 1$ **to** n **do**
        for $w = 1$ **to** d **do**
            if $w < H[i, j]$ **then**
                $M[i, j, w] = \underset{a \in \{-1,0,1\}}{Max} \{M[i-1, j+a, d]\}$
            else
                $M[i, j, w] = \underset{a,b \in \{-1,0,1\}}{Max} \{M[i-1, j+a, d], G[i, j] + M[i-1, j+b, d-H[i, j]]\}$
return M

The worst case running time is $\Theta(mnd)$. The three nested loops run for n, m, d iterations, with each iteration takes a constant time for random-access lookup in array M, H and possibly G. This is possible because at any iteration, $M[i-1, j, d]$ for all $j = 1 \cdots n$, $w = 1 \cdots d$ is already computed in the previous iteration of the outer most loop.

5. **Actual Solution**

`Drill-Gold` (d, H, G)
$M \leftarrow [0 \cdots m, 0 \cdots n, d]$
for $j = 0$ **to** n **do**
    for $w = 0$ **to** d **do**
        $M[0, j, w] \leftarrow 0$
for $i = 1$ **to** m **do**
    for $j = 1$ **to** n **do**
        for $w = 1$ **to** d **do**
            if $w < H[i, j]$ **then**
                $M[i, j, w] = \underset{a \in \{-1,0,1\}}{Max} \{M[i-1, j+a, d]\}$
            else
                $M[i, j, w] = \underset{a,b \in \{-1,0,1\}}{Max} \{M[i-1, j+a, d], G[i, j] + M[i-1, j+b, d-H[$
return M