Name: Chloe Hu
Student ID: 21044009
Email: chuap@connect.ust.hk

1.

a) To prove there exists a minimal cover A, first assume that A be a cover for I. Consider a point $x \in A$ where x is not in $\{f1,\ f2\ldots\ fn\}$.

Also I' = $\{I_{i,1}, I_{i,2}, I_{i,3} \ldots I_{i,k}\}$ is the set of intervals covered by x. So, x is found in $[S_{i,1}, S_{i,2}, \ldots S_{i,k}, F_{i,1}, F_{i,2}, \ldots F_{i,k}]$. The minimum F out of $[F_{i,1}, F_{i,2}, \ldots F_{i,k}]$, would cover all the intervals in I'. Consider $A' = A \cup \{f\} - \{x\}$. If an interval I is covered by A, then it is also covered by A'.

So, if we start with any minimal cover A, we can continue to replace any point that is not in $\{F_1, F_2 \ldots F_j\}$ with a point in $\{F_1, F_2 \ldots F_j\}$ to maintain the minimal cover.

b) Pseudocode:

```
// sort so that $F_1 < F_2 < F_3 \ldots < F_n$
Sort the intervals by finish times in increasing order
// initialise last to the first endtime to ensure this time is covered
Set last = $F_1$
Let A = {last}
// check if the interval is covered, if not add to A
Loop from $j$ = 2 to $n$
    If $s_j$> last                    // current interval isn't covered
        Set last = $F_j$
        Add last to $A$
```

Description: The algorithm first sorts by increasing finishing time. If the current interval is not covered yet, add the finish time to A. This ensures that A is a minimal cover since it contains the fewest possible points that cover all the intervals.

c) We will prove that the output generated by greedy is optimal using induction:
First let G be a cover generated by the pseudocode in part b). To prove this, if F_j is added to A, then I_j is trivially covered. If F_j is not added then some point has already been added to cover I_j. This point must be greater or equal to s_j.Thus, every I_j is covered by some point in A.

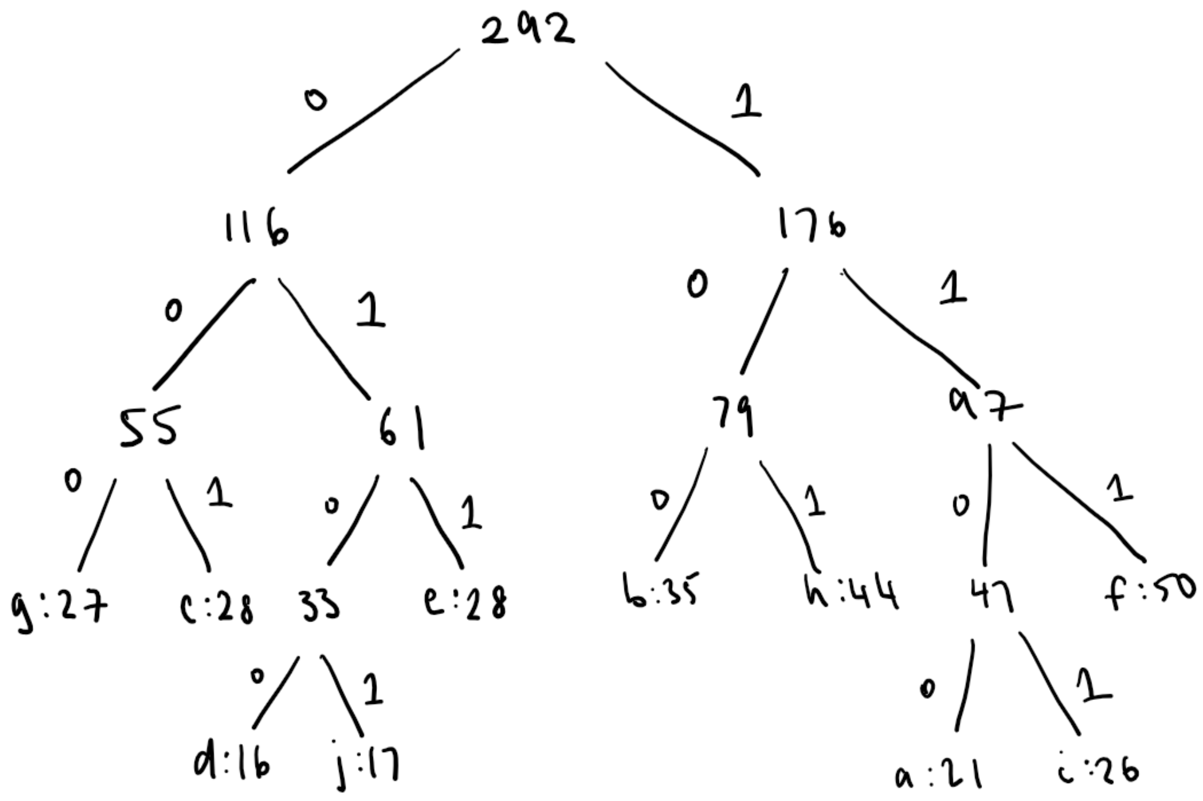
Now let O be an optimal, minimal cover. We assume that G is different to O. Let $a_1, a_2 \ldots a_k$ denote the set of intervals selected by greedy. Let $b_1, b_2 \ldots b_m$ denote the set of intervals selected by the optimal.

Now find the largest value r, where $a_1 = b_1, a_2 = b_2 \ldots a_r = b_r$ and $a_{r+1} \neq b_{r+1}$.
$\ldots a_r$. By the definition of the greedy algorithm $a_{r+1} < b_{r+1}$ since greedy always chooses the interval with the minimum finish time. As a result we can replace b_{r+1} in O with a_{r+1}. We can keep doing this until we reach the end of both solutions. At this point both solutions would be exactly the same where $G = O$. Thus the greedy solution is the optimal solution.

d) Sorting the intervals in ascending finish times uses $O(nlogn)$ in the worst case since we can sort in $O(nlogn)$ by using mergesort.
The loop iterates from $2\ to\ n$ times ($O(n)$) and does a maximum of one comparison and value assigning in $O(1)$ time in each iteration.
So the total runtime is $O(nlogn) + O(n) = O(nlogn)$.

2.

a) Huffman tree:



b) Final codeword

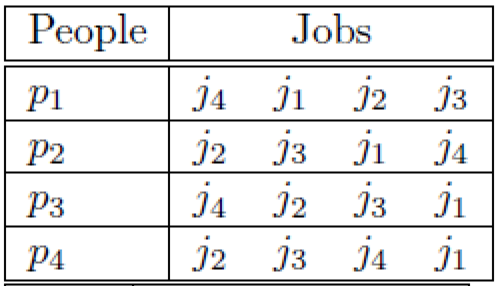
Letter	Codeword
a	1100
b	100
c	001
d	0010
e	011
f	111
g	000
h	101
i	1101
j	0101

c) Ordered codeword

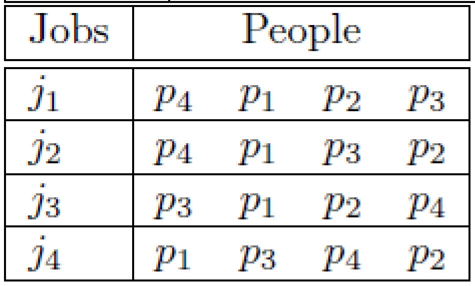
Letter	Codeword
b	100
c	001
e	011
f	111
g	000
h	101
a	1100
d	0010
i	1101
j	0101

3.

Match	Free	Rejects
(p1,j4)		
(p2,j2)		
(p3,j4)		j4 rejects p3
(p3,j2)	p2	
(p2,j3)		
(p4,j2)	p3	
(p3,j3)	p2	
(p2,j1)		



People	Jobs			
p_1	j_4	j_1	j_2	j_3
p_2	j_2	j_3	j_1	j_4
p_3	j_4	j_2	j_3	j_1
p_4	j_2	j_3	j_4	j_1



Jobs	People			
j_1	p_4	p_1	p_2	p_3
j_2	p_4	p_1	p_3	p_2
j_3	p_3	p_1	p_2	p_4
j_4	p_1	p_3	p_4	p_2

Bolded are the final pairings.

4.

Design DP algorithm:

Sort the tasks by deadline in ascending order in an array task[n]. The array contains elements 1 to n.

Initialise an array value[n] which holds the cumulative values for each index from 1 to n. Set value[0] = task[0].value.

For tasks from 1 to n, fill up value[n] in a bottom-up manner. This is accomplished by using binary search to find the index of the last task which does not conflict with the previous task ie. the start time of the prospective job is less than or equal to the given jobs finish time (which is calculated by summing the start time and length).

Then a dummy value is used to check whether the current value[i] or previous value[i-1] is larger. The max of the two values is stored in value[i].

Define and explain notation:
For a job “i”, we have [r(i), d(i), w(i), l(i)] which reference the release, deadline, weight and length respectively.

Let’s define p(j) as the largest index i < j such that task “j” is compatible with task “i”. The index p(j) is computed using binary search in O(logn) time.

Define and explain recurrence and BCs:
The boundary condition is when value[0] = 0 since if there is no tasks to schedule there is a value of 0.

The recurrence is value[j] = max{task[j].value + value[j - 1], value[j-1]}. This is because there are two options for a value[j], where it is either selected or not selected. If it is selected then tasks p(j) + 1, p(j) + 2… n cannot be used and the solution must contain an optimal solution to the subproblems 1, 2… p(j). If the task is not selected then the optimal solution to the problem must include a solution from 1,2… j - 1.

Pseudocode:

```
1. Sort jobs by deadlines so that d1<d2… dn
2. Int time = 0
3. For j = 1 to n
    a. If task[j].value + V[p(j)] > V[j-1] && time + task[j].length < task[j].deadline
        i. V[j] = task[j].value + V[p(j)]
        ii. time += task[j].length
        iii. Keep[j] = 1
    b. Else
        i. V[j] = V[j-1]
        ii. Keep [j] = 0
4. Let j = n
5. While j > 0
    a. If keep[j] = 1
        i. then print j
        ii. j = p(j)
    b. Else
        i. j = j -1
```

To find p(j): use binary search on the sorted task[n] array to find the largest compatible task where p(j).start > time and p(j).length + time < deadline.

Runtime:
Sorting the jobs by deadline can be done in O(nlogn) time with mergesort. The recursive step starting on line 3 iterates through all n tasks, thus takes O(n) time. The backtracking from line 4 takes O(n) time since, in the worst case, it iterates through all the tasks.

As a result the total run time of the algorithm is O(nlogn).

5.

Design DP algorithm:

Use DP to store and use the subproblems solved for previous indexes. We will do this by initialising a variable maxLength to keep track of the longest path. Then recursive call a depth first search, comparing if the current or previous path is longest.

Define and explain notation:

Def. maxPath[i][j] = the longest increasing path from the cell [i,j]

Define and explain recurrence and BCs:

The BC's

- When the matrix is of size 0, we return 0: maxPath[0][0] = 0.
- When the matrix has 1 value, we return 1: maxPath[i][j] = 1.

The recurrence is: maxPath[i][j] = max(maxLength, findMaxPath(newRow, newCol) which finds the maximum length by comparing the current maxLength with maxLengths found by the recursive call.

Pseudocode:

```
maxLength = 0
i_end = -1
j_end = -1
directions = [(-1, 0), (1, 0), (0, -1), (0, 1)]

findMaxPath(i, j):

  1. Let maxPath[0..n][0..n] and keep[0..n][0..n] be a new array of all 0        // initialise
  2. For i from 1 to n                                                    // recursive step
       a. For j from 1 to n                                               // max 4*O(n^2)
            i. For dx, dy in directions
                 1. x = i + dx
                 2. y = j + dy
                 3. if 0 <= x < n and 0 <= y < n and A[x][y] > A[i][j]
                      a. maxPath[x][y] = maxPath[i][j] + 1                 // O(1)
                      b. keep [x][y] = 1
                 4. Else
                      a. maxPath[x][y] = maxPath[i][j]                     // O(1)
                      b. keep[x][y] = 0
  3. maxLength = max(maxPath[i][j])                                       // max O(n^2)

// printing out the max increasing path
  4. Let count = maxLength
  5. Let finalPath = []
```

```
6. While count > 0                                   // print the indices
    a. For i from n to 1
        i. For j from n to 1
            1. If keep[i][j] = 1                     // O(1)
                a. Append [i][j] to finalPath
                b. count--

7. return maxLength, finalPath
```

Runtime:

The initialisation step costs O(n^2) as we must fill two n*n sized arrays with 0.
The recursive step costs O(n^2) as we iterate through n*n cells 4 times, since there are maximum of 4 directions the path can increase towards. Since 4 is a constant, the time complexity is O(n^2).
The printing step costs O(n^2) as it loops through all n*n cells and appends the indices to the finalPath array.

In conclusion the whole algorithm costs O(n^2) to find the magnitude of the longest increasing path and also output the indices of the final array.