**Question 1** (Huffman Code, 30 points). *Compute an optimal Huffman Code (or the corresponding tree) for the alphabet with the following letter frequencies: A - 5, B - 6, C - 7, D - 8, E - 14, F - 20, G - 40, H - 50, I - 80, J - 120.*

We use the standard Huffman Code algorithm. The two lightest nodes are A and B which we merge into K of weight 11. We then merge C and D into L of weight 15. We merge K and E into M of weight 25. We merge L and F into N of weight 35. We merge M and N into O of weight 60. We merge G and H into P of weight 90. We merge O and I into Q of weight 140. We merge P and J into R of weight 210. We merge R and Q into S of weight 350, and we are done. The tree is displayed below:

```
A 5  
    /  
   /    
B 6    C 7
    /    /  
   /    /    
D 8    E 14
       /    /  
      /    /    
     F 20  G 40
         /    /  
        /    /    
       I 80  J 120
         /    
        /      
       G 40    H 50
```

This could be turned into an encoding for example as:

- A - 00000
- B - 00001
- C - 00100
- D - 00101
- E - 0001
- F - 0011
- G - 100
- H - 101
- I - 1001
- J - 1011
- K - 0000
- L - 0010
- M - 0100
- N - 0101
- O - 0110
- P - 0111
- Q - 1000
- R - 1001
- S - 1010
H - 101
I - 01
J - 11
Question 2 (Simple Weights MST, 35 points). Let $G$ be a weighted, undirected graph with edge weights in \{1, 2, 3, \ldots, k\}. Give an $O(k(|V| + |E|))$ for computing the weight of a MST on $G$.

The algorithm is quite simple

Let $\text{tot} = |V| - 1$
For $w = 1 \ldots k-1$
    Let $G_w$ be the graph $G$ with only the edges of weight at most $w$
    $\text{tot} += \text{number of connected components in } G_w - 1$
Return $\text{tot}$

The runtime analysis is similarly easy. Each $G_w$ is computed in linear time as are its connected components. The final runtime is thus $O(k(|V| + |E|))$.

To show correctness note that running Kruskal’s algorithm on $G$ to compute the minimum spanning tree, we first add edges of weight 1 until we cannot add any more without creating a cycle. This happens when all of the connected components of $G_1$ are connected up. Therefore, there are $|V| - |CC(G_1)|$ edges of weight 1 in $T$. We then add edges of weight 2 until all the connected components of $G_2$ are completed. Thus, the total number of edges of weights 1 and 2 is $|V| - |CC(G_2)|$. Thus, the number of edges of weight 2 are $|CC(G_1)| - |CC(G_2)|$. Similarly, the number of edges of weight 3 in the minimum spanning tree is $|CC(G_3)| - |CC(G_2)|$ and so on. Thus the weight of the minimum spanning tree is

$$|V| - |CC(G_1)| + \sum_{w=2}^{k} w(|CC(G_{w-1})| - |CC(G_w)|)$$

$$= |V| + |CC(G_1)| + |CC(G_2)| + \ldots + |CC(G_{k-1})| - k|CC(G_k)|$$

$$= |V| + |CC(G_1)| + |CC(G_2)| + \ldots + |CC(G_{k-1})| - k$$

$$= (|V| - 1) + (|CC(G_1)| - 1) + (|CC(G_2)| - 1) + \ldots + (|CC(G_{k-1})| - 1),$$

which is what our algorithm computes.

Alternative Solution: Alternatively, we can proceed using a modification of Prim’s algorithm. In particular, our priority queue will only ever need to store numbers in \{1, 2, 3, \ldots, k, \infty\}. This can be implemented more efficiently by using an array of linked lists where the $i^{th}$ entry in the array stores a linked list of all the elements in the queue with key equal to $i$. It is easy to see that this data structure performs insert and decrease key operations in $O(1)$ time and delete min operations in $O(k)$ time. Prim’s algorithm needs to perform $|V|$ inserts and delete mins and at most $|E|$ decrease keys. This leads to a final runtime of $O(k(|V| + |E|))$.

This in fact can be improved further. If the set of non-empty bins in this array are stored in a Fibonacci Heap, a clever implementation can perform insert and decrease key operations in constant time, and delete min operations in $O(\log(k))$ time. This gives a final runtime of $O(\log(k)|V| + |E|)$.
**Question 3** (Tiling, 35 points). Let $R$ be a subset of a $3 \times n$ grid. Give an algorithm that determines whether or not $R$ can be tiled by $2 \times 1$ rectangles. See Figure 1 below for an example. For full credit, your algorithm should run in $O(n)$ time. You may assume that $R$ is presented by a $3 \times n$ array whose $(i, j)$-entry tells your algorithm whether or not the square $(i, j)$ is in $R$ or not.

![Figure 1: Here $R$ is a $3 \times 6$ grid with four squares removed tiled by seven $2 \times 1$ rectangles.](image)

We proceed by dynamic programming. We create an $n \times 8$ table ($8 = 2^3$). The entries of the table encode for each column $i$ and each subset of the squares of $R$ in this column whether or not it is possible to tile the rows to the right of $i$ along with the selected squares of row $i$. We fill in this table starting with large $i$ and working backwards. When $i = n$, we explicitly check to see if the pattern of squares in the last column can be tiled (a finite computation). For smaller values of $i$, consider all of the (finitely many) ways that dominoes could cover the selected tiles in column $i$. For each of them, see if the columns to the right of $i + 1$ along with the uncovered tiles in column $i$ can be successfully tiled (this will be a lookup in the table). If for any such case, it is possible, mark this entry as possible and otherwise mark it as impossible. Once this table has been filled out, the answer is just a corresponding table entry. There are $O(n)$ entries to this table, and each is filled out in constant time, so the runtime is $O(n)$.

To show correctness, we note that when filling in an entry we may assume that all previous entries of our table have been correctly filled out. Some configuration is tileable, if and only if there is a way to place our tiles to cover the leftmost column in such a way that the remainder of the region is tileable. Our algorithm checks exactly whether or not this is the case.