COMP 3711 Design and Analysis of Algorithms
Solutions to Assignment 3

1. Define $c[i]$ to be the length of the longest increasing subsequence that ends at x_i. Note that the length of the longest increasing subsequence in X is $\max_{1 \le i \le n} c[i]$. The longest increasing subsequence that ends with x_i has the form $< Z, x_i >$ where Z is the longest increasing subsequence that ends with x_r for some $r < i$ and $x_r \le x_i$. Thus, we have the following recurrence relation:

$$c[i] = \begin{cases} 1 & \text{if } i = 1 \\ 1 & \text{if } x_r > x_i \text{ for all } 1 \le r < i \\ \max_{\substack{1 \le r < i \\ x_r \le x_i}} c[r] + 1 & \text{if } i > 1 \end{cases}$$

We compute all the $c[i]$'s from $i = 1$ to n. Evaluating the recurrence takes $O(n)$ time, so the total running time is $O(n^2)$.

In order to report the optimal subsequence we need to store for each i, not only $c[i]$ but also the value of r which achieves the maximum in the recurrence relation. Denote this by $r[i]$. If $c[i]$ is a base case, we set $r[i] = 0$. Then we can trace the solution as follows. Let $c[k] = \max_{1 \le i \le n} c[i]$. Then x_k is the last number in the optimal subsequence. Then we update $k \leftarrow r[k]$. Now x_k is the second to last number. Then update $k \leftarrow r[k]$ again and repeat the process until $k = 0$.

2. This problem is similar to the 0-1 Knapsack problem, except that there is no "value" that we want to optimize. We just want to check if it possible to fully pack the knapsack. Thus we use a Boolean array, and define $B[i, w] = true$ if there is a subset of integers in $\{a_1, \dots, a_i\}$ that adds up to w. The recurrence is thus

$$B[i, w] = B[i-1, w] \text{ or } B[i-1, w-a_i].$$

The base case is $B[0, w] = false$ for any w, $B[i, w] = false$ for any $w < 0$, and $B[i, 0] = true$ for any i. We can compute all the $B[i, w]$'s from $i = 1$ to n, and for each i, we compute each $B[i, w]$ from $w = 1$ to W. The total running time is thus $O(nW)$. Finally, we return $B[n, W]$.

3. Define $L[i, j]$ to be the length of the longest palindromic subsequence for the substring $x[i, ..., j]$. If we look at $x[i, .., j]$, then we can find a palindrom of length at least 2 if $x[i] = x[j]$. If they are not same then we seek the longest palindromic subsequence in $x[i+1, ..., j]$ or $x[i, ..., j-1]$. Also every character $x[i]$ is a palindrom in itself of length 1. Therefore, we have the following recurrence:

$$L[i, j] = \begin{cases} L[i+1, j-1] + 2 & \text{if } x[i] = x[j] \\ \max\{L[i+1, j], L[i, j-1]\} & \text{otherwise} \end{cases}$$

$$\begin{aligned} L[i, i] &= 1 \ \forall i \in (1, ..., n) \\ L[i, i-1] &= 0 \ \forall i \in (2, ..., n) \end{aligned}$$

We compute all $L[i, j]$ from shorter strings to longer strings, similar to the optimal BST problem. More precisely, we first compute all $L[i, j]$ such that $j - i = 1$, then all $L[i, j]$

such that $j - i = 2, \ldots,$ until we have $A[1, n]$. It takes $O(1)$ time to compute each $L[i, j]$, so the total running time is $O(n^2)$.

To find the actual longest palindromic subsequence, we keep all the choices for each $L[i, j]$ in an array $r[i, j]$. Note that each $L[i, j]$ has been computed from one of 3 choices. Then we start from $r[1, n]$ and print out the palindrom using the following algorithm:

PRINT(L, r, i, j):
if $L[i, j] = 1$
    print $x[i]$
    return
if $r[i, j] = 1$
    print $x[i]$
    PRINT$(L, r, i + 1, j - 1)$
    print $x[j]$
if $r[i, j] = 2$
    PRINT$(L, r, i + 1, j)$
if $r[i, j] = 3$
    PRINT$(L, r, i, j - 1)$

4. The algorithm will be similar to BFS or DFS, except that we only visit vertices that are active. And whenever we visit a vertex, we spread its influence to its neighbors. The following code shows the BFS way of doing this by using a queue to store all the active vertices. You can also use a stack, which will make the algorithm more similar to DFS.

```
Q = empty
Q.enqueue(r)
count = 1
for each vertex v
    d(v) = 0   // d stores the total amount of influence received by v
    color(v) = WHITE
color(r) = BLACK
while (Q is not empty)
     v = Q.dequeue()
     for each u in Adj[v]
         if color(u) = WHITE then
             d(u) = d(u) + w(v, u)
             if d(u) >= t(u) then
                 color(u) = BLACK
                 Q.enqueue(u)
                 count = count + 1
return count
```

Since the algorithm does no more work than BFS, the running time is $O(V + E)$.