Name: Param Somane **Access ID:** pss5256

Problem 1 **Points:**

Let $A[1\cdots n]$ be the given array and k be such that $1 \leq k \leq n$. The longest subsequence of A that includes $A[k]$ is the concatenation of $opt(k)$ (which is the longest subsequence of $A[1\cdots k]$ with $A[k]$ being the last element using the definition from notes of lecture 25) with either the longest increasing subsequence of $A[k+1\cdots n]$ if $k < n$ or ϕ (empty array) if $k = n$. This holds because a subsequence of A containing $A[k]$, given by $(A[i_1], A[i_2], \cdots, A[i_{k-1}], A[i_k], A[i_{k+1}], \cdots, A[i_m])$ with $i_k = k$ and $1 \leq i_1 < i_2 < \cdots < i_k < \cdots < i_m \leq n$ is the longest (among all such subsequences) if and only if $(A[i_1], A[i_2], \cdots, A[i_{k-1}], A[i_k]) = opt(k)$ and $(A[i_{k+1}], \cdots, A[i_m])$ is the longest increasing subsequence of $A[k+1\cdots n]$. Recall that the longest-increasing-subsequence algorithm (from note 25) runs in $O(n^2)$ time. Thus, we have the following algorithm.

Algorithm Longest-Increasing-Subsequence-Ak ($A[1\cdots n]$, k $(1 \leq k \leq n)$)
- longest-increasing-sequence($A[1\cdots k]$); //Yields tracking-back pointers $T(i), \forall\, 1 \leq i \leq k$
- init $index = k$;
- init $subsequence1 = \phi$;
- init $subsequence2 = \phi$;
- init $subsequence1.append(A[k])$;
- while $T(index) \neq 0$
  - $subsequence1.append(A[T(index)])$;
  - $index = T(index)$;
- end while;
- reverse $subsequence1$ //Now $subsequence1 = (A[i_1], A[i_2], \cdots, A[i_k]) = opt(k)$
- longest-increasing-sequence($A[k+1\cdots n]$);
- /* The function call above yields tracking-back pointers $T'(i)$, and the length $L'(i)$ of $opt(i)$, $\forall\, k+1 \leq i \leq n$ */
- Find $L'(q) = \max_{k+1 \leq j \leq n} L'(j)$; //length of longest increasing subsequence of $A[k+1\cdots n]$
- init $index = q$;
- while $T'(index) \neq 0$
  - $subsequence2.append(A[T'(index)])$;
  - $index = T'(index)$;
- end while;
- reverse $subsequence2$ //Now $subsequence2 = (A[i_{k+1}], A[i_{k+2}], \cdots, A[i_m])$
- report $subsequence1$.concatenate($subsequence2$);
- /*$(A[i_1], A[i_2], \cdots, A[i_{k-1}], A[k], A[i_{k+1}], \cdots, A[i_m])$*/

end algorithm;

Running time: Since the algorithm longest-increasing-sequence runs in $O(m^2)$ time for an inputted array $B[1\cdots m]$, construction of the first and second subsequences above takes $O(k^2)$ and $O\left((n-k)^2\right)$ time respectively. Reversing of the sequences and their subsequent concatenation is carried out in linear time. Ergo, the algorithm runs in $O(k^2+(n-k)^2+n) = O(n^2)$ time.

Problem 2 **Points:**

(Incorrect algorithm) Let $d(A,B)$ represent the edit-distance between the two given strings A and B along the same lines as defined in lecture notes 26. Observe that the largest common subsequence between A and B is precisely the subsequence generated by all the $a_i's$ and $b_j's$ that 'match-up' in the optimal alignment of A and B corresponding to the edit-distance. This is because $d(A,B)$ corresponds to the minimization of characters in A and B that either do not match-up due to substitution operation or are omitted by virtue of insertion or deletion. Hence, we modify the edit-distance algorithm in note 26 to construct a table $P(i,j)$ of traceback pointers along with the dynamic programming table $F(i,j)$ that is constructed by the algorithm. This will allow us to find the longest common subsequence (LCS) explicitly. The following pseudo-code, which is a modified version of the edit-distance algorithm provided in the notes, illustrates this idea.

Algorithm Lowest-Common-Sequence ($A[1\cdots n] = a_1a_2\cdots a_n$, $B[1\cdots m] = b_1b_2\cdots b_m$)
- init $F(i,0) = i$, for any $0 \le i \le n$;
- init $P(i,j) = \phi$ (NULL), for any $0 \le i \le n$, $0 \le j \le m$;
- for $i = 1 \to n$
  - for $j = 1 \to m$
    - $F(i,j) = \min\{F(i-1,j-1) + \delta(X[i] \ne Y[j]), F(i-1,j)+1, F(i,j-1)+1\}$;
    - if $F(i,j) = F(i-1,j-1) + \delta(X[i] \ne Y[j]) : P(i,j) = (i-1,j-1)$;
    - else if $F(i,j) = F(i-1,j)+1 : P(i,j) = (i-1,j)$;
    - else: $P(i,j) = (i,j-1)$;
  - end for;
- end for;
- init $index = (n,m)$; //Corresponds to $F(n,m) = d(A,B)$
- /*We denote $index = (index[0], index[1])$*/
- init $LCS = \phi$;
- while $index \ne NULL$
  - if $A[index[0]] = B[index[1]]$
    - $LCS.append(index)$;
    - $index = P(i,j)$;
  - end if;
- end while;
- reverse LCS; //Now $LCS = ((i_1,j_1),(i_2,j_2),\cdots(i_k,j_k))$ and $A[i_l] = B[j_l]$, $\forall\, 1 \le l \le k$
- init $k = length(LCS)$;
- report LCS and k;

end algorithm;

Running time: Since the table of traceback pointers $P(i,j)$ is constructed simultaneously while forming the dynamic programming table $F(i,j)$ which stores the edit distance between $A[1\cdots i]$ and $B[1\cdots j]$, running time of the algorithm is the same as the edit-distance algorithm, that is $O(|A||B|) = O(mn)$.

Problem 3 **Points:**

It is assumed that the points for each problem x_i is non-negative. Our first goal is to divide the given problem into sub-problems such that the optimal substructure property is preserved. Define $H[1\cdots n]$ to be an array where $H[i]$ denotes the highest number of points that can be attained in the subspace $(i, i+1, i+2, \cdots, n)$ of all problems following the i^{th} problem, for each $1 \leq i \leq n$. If we elect to solve the i^{th} problem, then we get x_i points, along with the most number of points that can be attained by solving problems from $(i+p_i+1)^{th}$ problem onward. If we choose not to solve the i^{th} problem, then the most number of points that can be attained in the above subspace will be $H[i+1]$. The optimal ordering of the problems can then be obtained by iterating through H. Ergo, we have the following recurrence relation:

$$H[i] = \max \begin{cases} x_i + H[i+p_i+1] \\ H[i+1] \end{cases}$$

Note that the base case is $H[n] = x_n$ because only the n^{th} problem can be solved in the subspace (n); thus, we analyze the sub-problems in reverse order. We keep track of the problems supposed to be solved corresponding to each $H[i]$ using an array of binary lists $B[1\cdots n]$ (each $B[i]$ is an i-bit binary list; 0 denotes the problem with that index is supposed to be skipped while 1 denotes that the problem should be attempted). We seek $B[1]$ for which the search subspace consists of all the problems $(1, 2, \cdots, n)$. This idea is portrayed in the algorithm below.

```
Algorithm Highest-Total-Points (p_i, x_i (∀ 1 ≤ i ≤ n))
    init H[1···n+1] = [0]*n and B[1···n+1] = NULL*n;
    H[n] = x_n and B[n] = (1);
    Set p_i = min{p_i, n−i}, ∀ 1 ≤ i ≤ n; //To ensure that the array index is not out of bounds
    for i = n−1 → 1
        H[i] = max(x_i + H[i+p_i+1], H[i+1]);
        if H[i] = x_i + H[i+p_i+1] //This case is prioritized; if p_i = 0, we still attempt problem i
            B[i].append(1); //We attempt problem i
            B[i].concatenate([0]*p_i); //We don't attempt the following p_i problems
            B[i].concatenate(B[i+p_i+1]);
        else //H[i] = H[i+1]
            B[i].append(0); //We don't attempt problem i
            B[i].concatenate(B[i+1]);
        end if;
    end for;
    report B[1];
end algorithm;
```

Note that we define $H[n+1]$ and $B[n+1]$ to account for the boundary case when $H[n+1]$ is checked in the recursion because the n^{th} problem was skipped.

Running time: Since $H[i]$ and $B[i]$ are computed in constant time (assuming that concatenation and appending takes constant time for $B[i]$) for each i within the for loop, the algorithm clearly runs in $O(n)$ time.

Problem 4 **Points:**

We iterate through the given array of values $V[1\cdots n] = (v_1, v_2, \cdots, v_n)$ (Kim's preferences of the ingredients) with the goal of finding the sub-array $(v_i, v_{i+1}, \cdots, v_j)$ of this array that corresponds to the maximum sum $\sum_{k=i}^{j} v_k$ among all the continuous sub-arrays of V. To this end, we define two variables: *currentsum* to store the sum of values in the current sub-array being examined and *maxsum* to store the maximum sum among all the continuous sub-arrays examined until the i^{th} iteration ($1 \leq i \leq n$).

When $currentsum > 0$, it may possibly lead to an update of *maxsum*; thus, we proceed to the next iteration while continuing to examine the current sub-array. However, $currentsum \leq 0$ implies that this sub-array either will not lead to a subsequent update of *maxsum* or $maxsum \leq 0$ because all the v_i's turned out to be non-positive real numbers (in this latter case, the burger will contain a single, least disliked, ingredient). Therefore, if $currentsum \leq 0$, then we can reset it to zero and begin examining a new sub-array starting from the next index. Lastly, observe that at each iteration, we set $maxsum = \max\{maxsum, currentsum\}$. The following algorithm illustrates this idea.

```
Algorithm Most-Liked-Sum (v_i, (∀ 1 ≤ i ≤ n))
    init currentsum = 0;
    init maxsum = −∞;
    for i = 1 → n
        currentsum = currentsum + v_i;
        if maxsum < currentsum : maxsum = currentsum;
        if currentsum < 0 : currentsum = 0;
    end for;
    report maxsum;
end algorithm;
```

Running time: Since the algorithm comprises of a single for loop, in which all the computations are done in constant time, the algorithm runs in $O(n)$ time.

Problem 5 **Points:**

Let $a_1 a_2 \cdots a_i$ be an i-bit binary sequence (i.e. for each $1 \leq j \leq i$, $a_j \in \{0,1\}$), for each $1 \leq i \leq n$. Let $N[i]$ represent the number of i-bit sequences that contain no adjacent "1"s, for every i. Observe that for $i \geq 3$, if $a_i = 1$, then $a_{i-1} = 0$ and this implies that we have a total of $N[i-2]$ possible $(i-2)$-bit sequences that, when concatenated with "01", yield i-bit sequences that necessarily end in "01". On the other hand, if $a_i = 0$, then all of $a_1, a_2, \cdots, a_{i-1}$ are free over $\mathbb{Z}_2 = \{0,1\}$ and so we get $N[i-1]$ possible $(i-1)$-bit sequences that, when concatenated with "0", yield i-bit sequences that end in "0".

These are the only possibilities since a sequence cannot end in "11". Moreover, for $i \leq 2$, we have the base cases $N[1] = 2$ (since "0" and "1") are the only choices) and $N[2] = 3$ (since "00", "01", and "10" are the only choices). As a result, we have the following recurrence relation:

$$N[i] = \begin{cases} 2 & ,\text{if } i = 1 \\ 3 & ,\text{if } i = 2 \\ N[i-1] + N[i-2] & ,\text{if } i \geq 3 \end{cases}$$

We seek the value of $N[n]$, namely the number of n-bit sequences that contain no adjacent "1"s. Hence, we have the following pseudo-code.

Running time: Since the algorithm below comprises of a single for loop, and computations within it take place in constant time, the algorithm runs in $O(n)$ time.

```
Algorithm Chanting-Patterns (n ∈ ℕ)
    init N = ϕ; //|N| = n
    if n = 1 : report 2;
    if n = 2 : report 3;
    N[1] = 2;
    N[2] = 3;
    for i = 3 → n
        N[i] = N[i − 1] + N[i − 2];
    end for;
    report N[n];
end algorithm;
```

Bonus Problem

The most intuitive way to approach this problem is to cut the plank in a manner such that the plank is divided into almost equal halves at every cut. This will yield a total cost of a little over $k + \frac{k}{2} + \frac{k}{4} + \cdots$, which is clearly a lower bound for all the possible costs. However, this procedure is erroneous, as is evident by the following counterexample. Let $k = 10$ and let the array of positions of markers be $M = \{1,2,3,4,5,6\}$. By the above discussion, it behooves us to make the first cut at 5. However, observe that cutting at 5 splits the plank into the left and right halves, wherein the left half can be cut with a total cost of 5(cut at 3)+2(cut at 4)+3(cut at 2)+2(cut at 1)=\$12 and \$5 for the right part. Hence, it would take $10 + 12 + 5 = \$27$ to cut the entire plank. On the other hand, if we make our first cut at 6, then the right half doesn't cost anything and the left half costs 6(cut at 5)+3(cut at 3)+2(cut at 4)+3(cut at 2)+2(cut at 1)=\$16, thereby accumulating to a cost of $10 + 16 = \$26$ for cutting the entire plank, which is better than the \$27 obtained in the prior scenario. Thus, we now seek to develop a dynamic programming algorithm that can accomplish our goal. Let $M[1 \cdots n]$ be the sorted array of positive numbers indicating the position of markers. Define $F(a,b)$ to be the minimum cost of cutting the portion of the plank between a and b, where $a, b \in M \cup \{0,k\}$. Clearly, $F(0,k)$ is the required quantity, where k is the length of the entire wooden plank and we consider both '0' and 'k' as endpoint extensions of M. Clearly, we have $F(0,k) = \min_{m \in M}\{F(0,m) + F(m,k)\} + k$, where $\min_{m \in M}\{F(0,m) + F(m,k)\}$ represents the minimum cost of cutting the left and right portions of the plank resulting from the first cut. Generalizing this formula for any endpoints $m_1, m_2 \in M \cup \{0,k\}$

of the region $[m_1, m_2]$ (where $m_1 < m_2$) on the plank that we are examining, we provide the recursive formula

$$F(m_1, m_2) = \min_{m \in M;\, m_1 < m < m_2} \{F(m_1, m) + F(m, m_2)\} + (m_2 - m_1)$$

We first initialize $F(i, j) = \infty$ for all $i, j \in M \cup \{0, k\}$ and compute $\min_{m \in M;\, m_1 < m < m_2} \{F(m_1, m) + F(m, m_2)\}$ by iterating through all $m \in M$ between m_1 and m_2 via a for loop. Note that we first need to check if $F(i, j) = \infty$ (that is, it has not been updated) and only then compute its value. Moreover, this follows from the invariant at play here that $F(i, j)$ does not change after its value has been set. Lastly, we report $F(0, k)$ at the end as the minimum cost required to make all marked cuts.

Running time: Since the computing part of $F(m_1, m_2)$ takes constant time because it only references values from the dynamic programming table maintained for F, the running time is clearly $O\left((n+2)^2\right) = O(n^2)$ as we need to fill $(n+2)^2$ values in the table for F.