**CS141, Fall 2021, Assignment #3** **Due: 11:59pm, Wed 11/03, 2021**
**Name: Rishab Dudhia** **Student ID: 862141444**

# 1 Smartest Students (25 + 3pts)

(1) If the only restriction was the total weight $W$, the problem would be exactly like the 0/1 Knapsack problem we talked about in class. Since there is no restriction on how many students she can take, the only choice is to pick the $i^{th}$ student, so the subproblem would just be $f[i, k]$ with $k$ representing the remaining weight and $i$ representing the current choice of student. Then, $f[i, k]$ would represent the optimal smartness for total weight $k$ using the first $i$ students. Without the restriction, the total students she can take is allowed to be more than T, so the only consideration to be taken is whether to pick student $i$ or not.

(2) (a): $1 \leq i \leq N$
We know that we need to identify the boundary of $i$, and we know we can pick from a range of $N$ students. Therefore, $i$ can be any value from 1 to $N$, and if it is 0, we have reached a base case.
(b): $T$
We know we need to bound the values of $j$, and since we know that $0 \leq k \leq W$, we know that the weight dimension of the table is complete. The last remaining dimension would be the amount of students chosen, so the maximum value that $j$ could be is $T$.

(3)

$$f[i, j, k] = \max \begin{cases} f[i-1, j, k] \\ f[i-1, j+1, k-w_i] + s_i \end{cases}$$

$$f[0, j, k] = 0$$

$$f[i, j, k] = 0 \quad j > T$$

$$f[i, j, k] = 0 \quad k < w_i$$

The equation for $f[i, j, k]$ is given by deciding whether or not choosing student $i$ is part of the optimal solution. Therefore, we calculate the subproblem of not choosing student $i$, while looking at the next student and keeping the values of $j$ and $k$ the same. Then, we calculate the subproblem of choosing student $i$ which leads us to increment the value of $j$ since we are adding one student to the chosen $T$ students and we must also subtract the the weight of student $i$ from the value of $k$ since the remaining weight will decrease with student $i$ taking up space on the plane. Since we are choosing to add the student onto the plane, we must also add student $i$'s smartness to the result of the recursive call. After calculating the two situations of picking student $i$ or not, we choose the larger value to save in the 3-D table at $f[i, j, k]$.

(4)
```
int maxSmart (vector<int>& weights, vector<int>& smartness, vector<vector<vector<int>>>& f,
      int i, int j, int k, int T) {
    if f[i][j][k] is not -1 return f[i][j][k]
    return and save 0 to f[i][j][k] if i = 0, j > T, or k < items.at(i).first

    ans = maxSmart(weights, smartness, f, i-1, j, k, T)
    temp = maxSmart(weights, smartness, f, i-1, j+1, k - weights.at(i), T) + scores.at(i)

    if temp > ans then ans = temp
    f[i][j][k] = ans
    return ans
```

```
}
```

Begin with the call,

$$\text{int ans} = \text{maxSmart (weights, smartness, f, N, 0, W, T);}$$

in the main function. The first two parameters of the function are vectors whose $i^{th}$ elements represent the $i^{th}$ student's weight and smartness, respectively. The values in the 3-D table $f$ are initialized to $-1$ to represent that the value of subproblem $f[i,j,k]$ has not been calculated. Then we pass in the initial values for $i$, $j$, and $k$ as $N$, 0, and $W$, respectively. Then, the subproblem $f[N,0,W]$ represents the optimal smartness when choosing from all $N$ students, with $T-0$, or all $T$, seats available, and all $W$ pounds available on the plane. In the function, maxSmart(), if the value of the subproblem $f[i,j,k]$ has been calculated, it is returned. Otherwise, if a base case is reached, 0 is saved at $f[i][j][k]$ and returned. Otherwise, the first integer *ans* is calculated to represent the optimal smartness of not choosing student $i$. The next integer *temp* is calculated to represent the total smartness if student $i$ is chosen. The larger value is saved to the *ans* variable and saved at $f[i][j][k]$. Then it is returned as the maximum smartness achievable from the first $i$ students with $T-j$ seats available and $k$ pounds remaining on the plane.

(5) The asymptotic complexity of my algorithm is $\Theta(n^2)$. From the algorithm in part (4), we can see that for calculating the total smartness for each of the first $i$ students for $1 \leq i \leq n$, where $n$ is the number of students to be chosen from, there are $n$ calls to the function. Therefore, for the algorithm to calculate the best total smartness for $n$ students, the complexity would be $\Theta(n^2)$. Also, there are two calls to the maxSmart() function on item $i-1$ within the call to maxSmart() on item $i$. Then, the recursion tree must have a height of $n$ since there must be $n$ calls to the function to reach the base case. We also know that with 2 calls to the function, there will be $2n$ leaves of the recursion tree. Then, the complexity of the algorithm would be the height of the tree times the number of leaves. Then the complexity would be $\Theta(n \cdot 2n)$ which is simplified to $\Theta(n^2)$. This confirms that our algorithm runs in $\Theta(n^2)$ time.

(6) (Bonus)

## 2 Morse Code (25pts)

| | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| A (3) | .- | B (2) | -... | C (1) | -.-. | D (2) | -.. | E (3) | . | F (3) | ..-. |
| G (2) | --. | H (3) | .... | I (3) | .. | J (1) | .--- | K (2) | -.- | L (1) | .-.. |
| M (2) | -- | N (2) | -. | O (1) | --- | P (2) | .--. | Q (2) | --.- | R (2) | .-. |
| S (1) | ... | T (2) | - | U (1) | ..- | V (1) | ... | W (1) | .-- | X (2) | -..- |
| Y (2) | -.-- | Z (1) | --.. | | | | | | | | |

(1) Morse code for $UCR$ is ..- -.-..-. from the given table. Yes, there are many interpretations of the Morse code for $UCR$ given that there is no separator. For example, the code could represent $EACR$, $EPDN$, or $EETTETEETE$. The first two given representations of the code for $UCR$ have a low stroke count of 9.

(2)
```
int minStrokes(vector<pair<char, int>>& chars, vector<vector<int, int>>& vals, int
        currChar, int j) {

    if currChar is 0 return 0
    if vals[i][j] has been calculated, return vals[i][j]
    if j - currChar > 3 return 0
    int currStrokes = 0;
    for (int i = 0; i < chars.size(); ++i) {
        if (check(X, currChar, j, chars.at(i).first)) {
            currStrokes = chars.at(i).second
            break
        }
    }
    int ans = minStrokes(chars, vals, currChar-1, j)
    if currStrokes is still 0 {
        vals[i][j] = ans
        return ans
    }

    int temp = minStrokes(chars,vals, currChar-1, currChar-1) + currStrokes
    if (temp < ans || (ans == 0)) ans = temp
    vals[i][j] set to ans
    return ans
}
```

It is assumed that the function minStrokes() is initially called in the main function with the call,

int ans = minStrokes(chars, vals, X.size(), X.size());

The subproblem $s[i,j]$ represents the minimum amount of strokes from a decoded message from the first $j$ symbols while choosing to use the decoded representation of the substring $X[i,j]$ or not. Then the recurrence for the algorithm would look as:

$$s[i,j] = \min \begin{cases} s[i-1,j] \\ s[i-1,i-1] + s_{ij} \end{cases}$$

In the recurrence, the first subproblem moves the $i$ pointer to the symbol in front of the current symbol at $i$ in $X$, reprsenting choosing not to use the decoded character of $X[i,j]$ and instead look at a longer string to decode. Then, in the subproblem, the substring $X[i,j]$ is one character longer than the

substring $X[i, j]$ of the original problem showing that instead of decoding the original substring, we added a new value to the front of the substring. This will represent a different decoded character than what the original substring represented which means a different stroke number could be found. In the second subproblem, we are choosing to use the decoded character mapped to by the substring $X[i, j]$, so then we start the subproblem at the first symbol before the substring $X[i, j]$, or $i - 1$. We must also add $s_{ij}$, stroke value of decoded $X[i, j]$, to the result of this subproblem since we are choosing to use the decoded value of $X[i, j]$. After getting a value for the subproblem $s[i - 1, i - 1]$ (minimum number of strokes for the first $i - 1$ symbols), we add the value to $s_{ij}$ which completes the second subproblem. Basically, we are trying to see if we should use the decoded character mapped to by substring $X[i, j]$ with stroke count $s_{ij}$ or if we can achieve a lower count of strokes by not using the decoded character of $X[i, j]$ and instead use the decoded character of $X[i - 1, j]$.
The original call to the function passes in two vectors, chars and vals, respectively. The vals vector is a table that represents the minimum strokes achieved from the first $j$ symbols while choosing the last symbol represented by the substring $X[i, j]$. Before the initial call to the function, all places in vals are initialized to -1 to represent that a value has not been calculated. The vector chars is a vector of pairs with the first value of the pair being a char to represent the decoded character and the second value being the number of strokes for that character. Then we pass in the size of $X$ as both the $currChar$ and $j$ parameters, and when $currChar$ is 0, we have reached a boundary case to return 0. Here in the first call to the function, the size of substring $X[i, j]$ is just 1, so we are looking at the choice to decode the last symbol or to decrement the $i$ value to decode the last two symbols. The base cases ensure that the distance between $currChar$ and $j$ is always less than or equal to 4 places since 4 is the greatest length of any encoded value. After the base cases, we also check to see if the current substring $X[i, j]$ actually represents a decoded value. If not, we can only look at the first subproblem of the recurrence since the second subproblem requires that the substring $X[i, j]$ is an encoded value.

(3) The time complexity of my algorithm would be $\Theta(n^2)$. Let $n$ be the size of the input string $X$. Then, while calculating the minimum stroke count for each substring of the first $i$ symbols for $1 \leq i \leq n$, there there are $n$ calls to the function. Therefore, for each $n$ number of symbols, there are $n$ calls to the function, so the function has complexity $\Theta(n^2)$.

(4) The minimum amount of strokes we can get from the sequence - -..- -. with characters:

```
A (3): .-
C (1): -.-.
E (3): .
G (2): --.
K (2): -.-
T (2): -
```

is 7 strokes. We can achieve this by encoding the string $GEG$.

# 3 Knapsack Algorithms can be Simple I (Bonus, 3pts + 1 candy)

# 4 Knapsack Algorithms can be Simple II (Bonus, 3pts + 1 candy)