CSE 202 Homework 2
Combinatorial Search and Optimization:
Dynammic Programming and Greedy Algorithms

1. **Single Word Reusable Boggle (SWoRB)**

   Problem You are given an $n \times n$ matrix of letters from a finite alphabet Σ, and a target word w in Σ of length m. You want to determine whether there is a (not necessarily simple, *i.e.* letters can be reused) path in the grid so that the letters along that path are w. Each step in the path can go from a point in the grid to any of its 8 neighboring points, including diagonal moves. Give an efficient algorithm to play SWoRB.

   Solution The important thing to note when spelling a word on the SWoRB matrix is that the exact path history is not important, only your position on the matrix and how far into the word you have gone. So, intuitively, we only need maintain information on the polynomially many *matrix position* $\times$ *word position* subproblems. The subproblems will be exactly those: an $n \times n \times m$ array, S, with element $S(i,j,k)$ indicating if it is possible to spell word $\{w_1 \ldots w_k\}$ ending at position (i,j) on the matrix.

   The computation begins by initializing the first plane of the matrix $\{S(i,j,1) : i,j \leq n\}$ with true if $M(i,j) = w_1$, false otherwise. When all elements $S(.,.,k)$ are filled in, elements of $\{S(i,j,k+1) : i,j \leq n\}$ are set to $true$ if $M(i,j) = w_{k+1}$ and any of the neighbors $\{S(i',j',k) : i' \in \{i-1,i,i+1\}, j' \in \{j-1,j,j+1\}, i' \neq i \vee j' \neq j\}$ are set to true. The order of evaluation within a plane does not matter (*i.e.*any $S(i,j,k)$ may be evaluted before any other $S(i',j',k)$), but all $S(.,.,k)$ must be evaluated before proceeding to $S(.,.,k+1)$. The answer to the problem is yes if any of the elements $S(.,.,m)$ are true.

   There are $n \times n \times m$ elements in this subproblem array, and each requires a constant number of comparisons or lookups ($\leq$ 9) to determine its value. So, the running time is $O(n^2 m)$. If we wanted to generate an example word path, we could pick any ending element, (i,j), such that $S(i,j,m)$ is true, and proceed backwards selecting any true neighbor on the previous plane. This could not be done in a forward direction, since a $true$ at position $S(i,j,k)$ does not guarantee that this word path will continue.

   Note: A similar problem is that of *counting* the number of paths through M that form a word w. This could be accomplished by a similar algorithm that used the same subproblems, but stored a count of successful paths to a particular point instead of simply if one existed or not. The final step would sum over all counts on the final plane. This could be further extended to randomly sample from the paths uniformly by selecting each backward step with probability proportional to its share of the paths.

2. Problem 16-3 from text (p. 325)

   Let c_{copy}, c_{delete}, $c_{replace}$, c_{insert}, $c_{twiddle}$ and c_{kill} be the costs of the operations. Let $ED(s_1 s_2..s_k, t_1 t_2..t_l)$ be the edit distance from s to t, and let ED_{op} be the edit distance given that we perform operation *op* first. Then we get the recursive definitions:

   (a) $ED(s,t) = 0 if k = l = 0, min_{op} ED_{op}(s,t) otherwise$,

   (b) $ED_{delete}(s_1..s_k, t_1..t_l) = c_{delete} + ED(s_2, ..s_k, t_1..t_l)$ (if $k > 0$, infinity otherwise);

   (c) $ED_{replace}(s_1..s_k, t_1..t_l) = c_{replace} + ED(s_2..s_k, t_2..t_l)$ if $k, l > 0$, infinity otherwise. (Note: this is going from the use of replace in their example, where the replaced symbol is automatically copied to the target and hence must be t_1. Other interpretations make the algorithm rather more complicated.)

   (d) $ED_{copy}(s,t) = c_{copy} + ED(s_2..s_k, t_2..t_l)$ if $s_1 = t_1$, infinity otherwise.

   (e) $ED_{insert}(s,t) = c_{insert} + ED(s_1..s_k, t_2..t_l)$ if $l > 0$, infinity otherwise.

   (f) $ED_{twiddle}(s,t) = c_{twiddle} + ED(s_3..s_k, t_3..t_l)$ if $s_1 = t_2$ and $t_1 = s_2$, infinity otherwise.

   (g) $ED_{kill}(s,t) = c_{kill}$ if $l = 0$, infinity otherwise.

   Note that the recursive calls in this definition are all to suffixes of the two strings, and that one of the suffixes is always proper. This leads to the following dynammic programming algorithm. Make an array of entries, $A(i,j)$, $1 \leq i \leq k+1$, $0 \leq j \leq l+1$, where each entry has two fields, the first representing the edit distance $ED(s_i..s_k, t_j..t_l)$, the second representing the first operation in a sequence that achieves that edit distance. (In the cases $i = k+1$ or $j = l+1$, the distance is the edit distance to or from the empty string respectively.) Fill

in the table according to the above recursion in the order: fill in the rows from $j = l + 1$ to $j = 1$, and fill in the columns in each row from $i = k+1$ to $i = 1$. Then the edit distance is the first field at $A(1, 1)$, and the sequence of operations can be found by tracing forward via the second fields. The size of the array is $O(kl)$ and the time to fill in any one entry requires doing $O(1)$ probes into the table, $O(1)$ additions, and finding the minimum of $O(1)$ numbers, so is constant time per entry. So the total memory and time are both $O(kl)$

3. Give the best algorithm you can for the following problem:

   Name Blackjack Hand Card Counting

   Instance An array A of n positive integers (cards with face values) with values from 1 to k, and positive integers $l < n, v < kn$.

   Problem Count the number of sets of l array positions (hands of l cards) whose total value is equal to v.

   Analyze your algorithm in terms of n, k and l. Your algorithm should take time polynomial in all 3 parameters.

   First, if $v > nl$, we can output 0 as the number of hands. Let $H[I, L, V]$ be the number of hands from $A[I], ..A[n]$ using L cards totalling V. (Define $H[I, 0, 0] = 1$, and otherwise $H[I, L, V] = 0$ unless both L, V are positive.) Then there are two ways of forming such hands: either we keep $A[I]$ and pick a hand of $L - 1$ cards summing to $V - A[I]$ or we pick the hand from $A[I + 1], ..A[n]$. This gives the recursion $H[I, L, V] = H[I + 1, L - 1, V - A[I]] + H[I + 1, L, V]$ provided $I < n$. If $I = n$, $H[n, L, V] = 1$ if $L = 1$ and $A[I] = A[n] = V$ (or if $L = V = 0$) and 0 otherwise. The recursive definition always increases I, so we want to fill it in the array H from $I = n$ to $I = 1$, and we can fill in L and V in any order. So our algorithm would first fill in , for $0 \leq L \leq l$ and $0 \leq V \leq v$, the entry $H(n, L, V)$ according to the rule for $I = n$. It would then, for $I = n - 1$ to 1, $0 \leq L \leq l$, $0 \leq V \leq v$, fill in $H(I, L, V)$ according to the recursive definition, The program then returns $H(1, l, v)$. Since the recursion is constant time, the time is proportional to the size of the array H, $O(nlv) = O(nl^2k)$ since $v \leq lk$.