Homework 6

CS/ECE 374 B

Due 8 p.m. **Thursday**, October 31

- Be sure to follow the dynamic programming steps described in the textbook / labs
- Look at the solved problem and rubric at the end. Be sure to include a description of the recurrence you are computing, including an English description of each subproblem
- Unlike section A, we do expect you to write an iterative implementation of your DP algorithm
- The problem descriptions generally use 1-based arrays, but you can use 0-based formulation in your problem, especially if you are using Python; just make a note that this is how your solution should be read.

1. *(This problem is essentially problem 1 in the textbook; you may find its problem description clearer / more amusing.)*

   A currency has bills in the following denomination: 1, 4, 7, 13, 28, 52, 91, 365. Your goal is to design an algorithm for finding the smallest number of bills that can be used for a given amount of currency.

(3) (a) Describe a recursive backtracking algorithm for counting the smallest number of bills that is needed to produce a given amount of currency. E.g., 90 can be represented as one 52 bill, one 28 bill, one 7 bill, and three 1 bills, for a total of 6. Do not make this algorithm efficient or analyze it.

Solution:

```
def make_change_rec(target):
    if target == 0:
        return 0
    else:
        # try all possible choices for the next bill and return the minimum
        # the last clause skips values that would result in a negative target
        return min(1+make_change_rec2(target−bill)
                   for bill in bills
                   if target−bill >= 0)
```

■

(7) (b) Design and analyze a dynamic programming algorithm for finding the smallest number of bills.

Solution: Our memoization array is going to be best$[i]$ representing the smallest number of bills that are needed to get the amount i. It follows the recurrence defined above, also stated as:

$$\text{best}[0] = 0$$
$$\text{best}[i] = \min_{b \in \text{bills}, b \le i} \text{best}[i-b]$$

We can see that elements of best depend only on smaller elements, so evaluating the recurrence starting at 1 suffices. This gives us the solution:

```
def make_change_dyn2(target):
    best = [0 for n in range(target+1)]
    for value in range(1,target+1):
        best[value] = min(1+best[value−bill] for bill in bills if value−bill >= 0)
    return best[target]
```

This problem takes $\Theta(n)$ time since the **min** operation is done over a constant number of bills. ■

(1 (bonus)) (c) What is the smallest number of bills where the greedy approach does not result in the smallest number of bills? The greedy approach repeatedly takes the largest bill that does not exceed the remaining amount. For full credit, list the greedy solution and the smaller non-greedy answer. Hint: you will likely want to use a computer program to solve this question.

Solution:

$$416 = 365 + 28 + 13 + 7 + 1 + 1 + 1 = 91 \times 4 + 52$$

■

(10) 2. *(This problem is essentially problem 11 in the textbook; you may find its problem description clearer.)*

You are given a list of n words and their typeset lengths, $\ell(1), \ldots, \ell(n)$. Your goal is to break them into lines of length L while minimizing slop. Each line has a maximum length of L and must contain at least one unit of whitespace between each word. Therefore, to put words i through j (inclusive) on a line, we must satisfy the following constraint:

$$(j-i) + \sum_{k=i}^{j} \ell(k) \leq L \tag{1}$$

The slop of a line is defined as the cube of the additional whitespace that is added on each line beyond what is required, i.e.:

$$\left(L - (j-i) - \sum_{k=i}^{j} \ell(k)\right)^3$$

Design an and analyze an efficient algorithm to break a list of words into lines while minimizing the total slop, i.e., the sum of slop on each line except the last. (The last line must still satisfy the constraint (1).) Your algorithm will be given a list of word lengths and will output the minimum slop achieved.

Solution: We can define $\text{minslop}[k]$ to be the minimum slop of the first k words being laid out that does count the slop on the last line. It is easy to see that it follows the following recurrence:

$$\begin{aligned}\text{minslop}[0] &= 0\\ \text{minslop}[i] &= \min_{0 \leq j < i, i-j-1+\sum_{k=j+1}^{i} \ell(k) \leq L} \text{minslop}[j] + \text{slop}(j+1, i)\end{aligned}$$

where slop is defined as above.

Again, it is easy to see that the array can be filled in starting from 0 in increasing order. To get the answer that ignores the slop on the last line, we will want:

$$\min_{0 \leq j < n, n-j-1+\sum_{k=j+1}^{n} \ell(k) \leq L} \text{minslop}[j]$$

Giving us the solution:

```
def typeset(l):
    # minslop[i]: minimum slop for typesetting
    # words l[:i] while counting the last line
    minslop = [0 for _ in range(len(l))]
    for i in range(1,len(l)):
        # keep track of the (minimum) length of
        # the last few words. Start at -1 to
        # account for one less space than # of words
        length = -1
        min_so_far = float("inf")
        for j in range(i-1,-1,-1):
```

```
            length += l[j] + 1
            if length > L:
                # line too long
                break
            min_so_far = min(min_so_far,minslop[j] + (L-length)**3)
        minslop[i] = min_so_far
    # find the actual result
    length = -1
    min_so_far = float("inf")
    for j in range(len(l)-1,-1,-1):
        length += l[j] + 1
        if length > L:
            break
        min_so_far = min(min_so_far,minslop[j])
    return min_so_far
```

Note there are two nested loops which have $O(n)$ iterations so the algorithm runs in $\Theta(n^2)$ time. ■

(10) 3. The edit distance between strings S and T is the minimum number of character insertions, deletions, and substitutions needed to change string S into string T. (This distance is described in detail in §3.7 of the textbook, where a dynamic programming algorithm for it is also discussed.)

In some cases, we only care about edit distance if it is small. E.g., if we want to find whether a misspelled word is similar to a dictionary word, we only want to consider words that are at a distance of at most 5. So given S and T, and a bound B, we want a function that returns either the edit distance between S and T, if it is less than or equal to B, or $+\infty$ (**float**("inf") in Python) if the edit distance is larger than B. So for example:

```
BoundedEditDistance("food","money", 5) == 4 # edit dist is 4
BoundedEditDistance("algorithm", "altruistic", 5) == float("inf")
    # real edit distance is 6
```

(3) (a) Consider the following recursive implementation of computing the edit distance. (It directly follows the recurrence defined in §3.7 of the text.)

```
def edit_recursive(S, T):
    """ Computes the minimum cost to edit string S to
    obtain string T. Character deletions, insertions,
    and substitutions all have a cost of 1 """
    if len(S) == 0:
        # insert all characters in T
        return len(T)
    if len(T) == 0:
        # delete all characters in S
        return len(S)
    # cost to delete one char of S
    del_cost = edit_recursive(S[:-1], T) + 1
    # cost to insert one char of T
    ins_cost = edit_recursive(S, T[:-1]) + 1
    if S[-1] == T[-1]:
        # zero cost to match chars
        match_cost = edit_recursive(S[:-1], T[:-1])
    else:
        match_cost = edit_recursive(S[:-1], T[:-1]) + 1
    return min(del_cost, ins_cost, match_cost)
```

Modify this implementation to compute the bounded edit distance. Your modification should be minor; in particular you should not use dynamic programming or memoization. But your modified algorithm

should run faster than the original one, at least in some cases. Do not analyze the algorithm's runtime complexity.

Solution:

```
def edit_recursive(S, T, B):
    """ Computes the minimum cost to edit string S to
    obtain string T. Character deletions, insertions,
    and substitutions all have a cost of 1. Return
    the cost, or 'float("inf")' if cost is > B """
    # if we've exceeded the bound, return infinity
    if B < 0:
        return float("inf")
    if len(S) == 0:
        # insert all characters in T
        if len(T) > B:
            return float("inf")
        else:
            return len(T)
    if len(T) == 0:
        # delete all characters in S
        if len(S) > B:
            return float("inf")
        else:
            return len(S)
    # cost to delete one char of S
    del_cost = edit_recursive(S[:-1], T, B-1) + 1
    # cost to insert one char of T
    ins_cost = edit_recursive(S, T[:-1], B-1) + 1
    if S[-1] == T[-1]:
        # zero cost to match chars
        match_cost = edit_recursive(S[:-1], T[:-1], B)
    else:
        match_cost = edit_recursive(S[:-1], T[:-1], B-1) + 1
    return min(del_cost, ins_cost, match_cost)
```

■

(7) (b) Design and analyze a dynamic programming algorithm for bounded edit distance. Your algorithm should be $o(n^2)$ when run on two strings of length n, assuming that the bound is $o(n)$.

Solution: The standard DP algorithm for edit distance (described in the textbook) computes the function $\mathsf{Edit}(i,j)$ that denotes the minimum edit distance between $S[0:i]$ and $T[0:j]$ in $\Theta(n^2)$ time. Our observation is that $\mathsf{Edit}(i,j) \geq |i-j|$. Therefore, we only need to consider the entries in $\mathsf{Edit}(i,j)$ that are at most B away from the diagonal.[1]

To represent this in a data structure, we define a $n \times 2B+3$ array BEdit where $\mathsf{BEdit}[i][j] = \mathsf{Edit}(i, i+j-b-1)$. Then we need to find $\mathsf{BEdit}[n][0]$.

[1] In fact for equal length strings we only really need to consider entries $B/2$ away from the diagonal, because the $B/2$ deletions (insertions) would need to later be used by $B/2$ insertions (deletions), but that is only a constant-time improvement.

Adjusting the recurrence from the text, we have:

$$
\begin{aligned}
BEdit[i][j] &= \infty && \text{for } i+j-b-1<0\\
BEdit[i][j] &= 0 && \text{for } i=0\\
BEdit[i][j] &= 0 && \text{for } i=b+1-j\\
BEdit[i][j] &= \infty && \text{for } j=0 \text{ or } j=2b+2\\
BEdit[i][j] &= \mathsf{clip}\,(\min\{\mathsf{BEdit}[i][j-1]+1, \mathsf{BEdit}[i-1][j+1]+1 \\
&\qquad \mathsf{BEdit}[i-1][j]+[S[i]\neq T[i]]) && \text{otherwise}
\end{aligned}
$$

where:

$$
\begin{aligned}
\mathsf{clip}(x) &= x && \text{if } x \le B\\
\mathsf{clip}(x) &= \infty && \text{otherwise}
\end{aligned}
$$

Note that, after filling the base cases, filling this array in row major order will ensure that the dependencies are met. This will take $\Theta(B \times n)$ time.

[code to be posted] ■

Solved Problem

4. A shuffle of two strings X and Y is formed by interspersing the characters into a new string, keeping the characters of X and Y in the same order. For example, the string BANANAANANAS is a shuffle of the strings BANANA and ANANAS in several different ways.

$$\texttt{BANANA}_{\texttt{ANANAS}} \qquad \texttt{BAN}_{\texttt{ANA}}\texttt{ANA}_{\texttt{NAS}} \qquad \texttt{B}_{\texttt{AN}}\texttt{AN}_{\texttt{A}}\texttt{A}_{\texttt{NA}}\texttt{NA}_{\texttt{S}}$$

Similarly, the strings PRODGYRNAMAMMIINCG and DYPRONGARMAMMICING are both shuffles of DYNAMIC and PROGRAMMING:

$$\texttt{PRO}^{\texttt{D}}\texttt{G}^{\texttt{Y}}\texttt{R}^{\texttt{NAM}}\texttt{AMMI}^{\texttt{I}}\texttt{N}^{\texttt{C}}\texttt{G} \qquad \texttt{DY}_{\texttt{PRO}}\texttt{N}_{\texttt{G}}\texttt{A}_{\texttt{R}}\texttt{M}_{\texttt{AMM}}\texttt{IC}_{\texttt{ING}}$$

Given three strings $A[1..m]$, $B[1..n]$, and $C[1..m+n]$, describe and analyze an algorithm to determine whether C is a shuffle of A and B.

Solution: We define a boolean function $\underline{Shuf}(i,j)$, which is TRUE if and only if the prefix $C[1..i+j]$ is a shuffle of the prefixes $A[1..i]$ and $B[1..j]$. This function satisfies the following recurrence:

$$
\underline{Shuf}(i,j) = \begin{cases}
\text{TRUE} & \text{if } i=j=0\\
\underline{Shuf}(0,j-1) \wedge (B[j]=C[j]) & \text{if } i=0 \text{ and } j>0\\
\underline{Shuf}(i-1,0) \wedge (A[i]=C[i]) & \text{if } i>0 \text{ and } j=0\\
\Big(\underline{Shuf}(i-1,j) \wedge (A[i]=C[i+j])\Big) & \\
\qquad \vee \Big(\underline{Shuf}(i,j-1) \wedge (B[j]=C[i+j])\Big) & \text{if } i>0 \text{ and } j>0
\end{cases}
$$

We need to compute $\underline{Shuf}(m,n)$.

We can memoize all function values into a two-dimensional array $\underline{Shuf}[0..m][0..n]$. Each array entry $\underline{Shuf}[i,j]$ depends only on the entries immediately below and immediately to the right: $\underline{Shuf}[i-1,j]$ and $\underline{Shuf}[i,j-1]$. Thus, we can fill the array in standard row-major order. The original recurrence gives us the following implementation. (Note the slight adjustments by 1 because Python uses 0-based arrays, unlike the description above.)

```
def shuffle(A, B, C):
    """ assumes len(C) = len(A) + len(B) """
    # initialize array to undefined
    shuf = [[None for _ in range(len(B)+1)] for _ in range(len(A)+1)]
    shuf[0][0] = True
    for j in range(1, len(B)+1):
        shuf[0][j] = shuf[0][j-1] and (B[j-1] == C[j-1])
    for i in range(1, len(A)+1):
        shuf[i][0] = shuf[i-1][0] and (A[i-1] == C[i-1])
        for j in range(1, len(B)+1):
            shuf[i][j]= False
            if A[i-1] == C[i+j-1]:
                shuf[i][j] = shuf[i][j] or shuf[i-1][j]
            if B[j-1] == C[i+j-1]
                shuf[i][j] = shuf[i][j] or shuf[i][j-1]
    return shuf[len(A)][len(B)]
```

The algorithm runs in $\underline{O(mn) \text{ time}}$. ■

Rubric: Max 10 points: Standard dynamic programming rubric. No proofs required. Max 7 points for a slower polynomial-time algorithm; scale partial credit accordingly.

Rubric: Standard dynamic programming rubric For problems worth 10 poins:

- 6 points for a correct recurrence, described either using mathematical notation or as (pseudo)code for a recursive algorithm.
  - \+ 1 point for a clear **English** description of the function you are trying to evaluate. (Otherwise, we don't even know what you're trying to do.) **Automatic zero if the English description is missing.**
  - \+ 1 point for stating how to call your function to get the final answer.
  - \+ 1 point for base case(s). −½ for one minor bug, like a typo or an off-by-one error.
  - \+ 3 points for recursive case(s). −1 for each minor bug, like a typo or an off-by-one error. **No credit for the rest of the problem if the recursive case(s) are incorrect.**
- 4 points for the dynamic programming algorithm
  - \+ 1 point for setting up the memoization data structure and dealing with base cases
  - \+ 2 points for correct evaluation order
  - \+ 1 point for time analysis
- It is not necessary to state a space bound.
- For problems that ask for an algorithm that computes an optimal structure—such as a subset, partition, subsequence, or tree—an algorithm that computes only the value or cost of the optimal structure is sufficient for full credit, unless the problem says otherwise.
- Official solutions will provide target time bounds. Algorithms that are faster than this target are worth more points; slower algorithms are worth fewer points, typically by 2 or 3 points (out of 10) for each factor of n. Partial credit is scaled to the new maximum score, and all points above 10 are recorded as extra credit.

  We rarely include these target time bounds in the actual questions, because when we have included them, significantly more students turned in algorithms that meet the target time bound but didn't work (earning 0/10) instead of correct algorithms that are slower than the target time bound (earning 8/10).