CS 124 Homework 4: Spring 2021

Your name: Sean Ty

Collaborators:

No. of late days used on previous psets: 0
No. of late days used after including this pset: 0

Homework is due Wednesday 2021-03-10 at 11:59pm ET. You are allowed up to **twelve** (college)/**forty** (extension school) late days through the semester, but the number of late days you take on each assignment must be a nonnegative integer at most **two** (college)/**four** (extension school).

Try to make your answers as clear and concise as possible; style will count in your grades. Be sure to read and know the collaboration policy in the course syllabus. Assignments must be submitted in pdf format on Gradescope. If you do assignments by hand, you will need to scan in your results to turn them in.

For all homework problems where you are asked to design or give an algorithm, you must prove the correctness of your algorithm and prove the best upper bound that you can give for the running time. Generally better running times will get better credit; generally exponential time algorithms (unless specifically asked for) will receive no or little credit. You should always write a clear informal description of your algorithm in English. You may also write pseudocode if you feel your informal explanation requires more precision and detail, but keep in mind pseudocode does NOT substitute for an explanation. Answers that consist solely of pseudocode will receive little or not credit. Again, try to make your answers clear and concise.

1. (a) **(20 points)** A challenge that arises in databases is how to summarize data in easy-to-display formats, such as a histogram. A problem in this context is the minimal imbalance problem. Suppose we have an array A containing n numbers, all positive, and another input k. Consider k indices $j_1, j_2, \ldots j_k$ that partition the array into $k+1$ subarrays $A[1, j_1], A[j_1+1, j_2], \ldots, A[j_k+1, n]$. The weight $w(i)$ of the ith subarray is the sum of its entries. The *imbalance* of the partition is

$$\max_i \left| w(i) - \left(\sum_{\ell=1}^{n} A[\ell] \right) / (k+1) \right|.$$

That is, the imbalance is the maximum deviation any partition has from the average size.

Give an algorithm for determining the partition with the minimal imbalance given A, n, and k. (This corresponds to finding a histogram with k breaking points, giving $k+1$ bars, as close to equal as possible, in some sense.)

For convenience, we let $c = \frac{1}{k+1} \sum_{j=1}^{n} A[j]$ be the "average." Also, we construct the variables (in say an array) $a_{i,j}$ such that $a_{ij} = \sum_{k=i}^{j} A[k]$. This can clearly be done in $O(n^2)$ time and space.

We then construct a 2D array $B[i, j]$ such that $0 \leq i \leq n$ and $0 \leq j \leq k$, and each element $B[i, j]$ is an ordered pair of a number and a list. We aim to make $B[i, j]$ correspond to the minimum imbalance (for the first term) and the partition list (for the second term). To do this, we initialize $B[i, 0] = (|a_{1,i} - c|, [])$, so that it corresponds to the empty partition of the first i elements of A. We then compute the other values of B through the following algorithm:

```
for j in range(1, k+1):
    for i in range(0, n+1):
        minVal = float("inf")
        indices = []
        for k in range(0, i+1):
            ## bash all k and look for best partition for j
            ## bashing based on the "size" of last block
            temp = max(B[k, j-1][0], |a_{k+1, i} -c|)
            if temp < minVal:
                ## update and stuff
                minVal = temp
                ## appending k since this means this k is
                ## "best" for the largest block
                indices = B[k, j-1][1].append(k)
        B[i, j][0] = minVal
        B[i, j][1] = indices
```

We claim that each $B[i, j][0]$ corresponds to the minimum imbalance and that each $B[i, j][1]$ corresponds to the partition that achieves this. We proceed by induction on j. This is vacuously true when $j = 0$, since the only way to split is the entire array itself. Now suppose that this result is true for all $k < j$, and we will prove the result for j.

Now fix some $i \in [0, n]$. We do casework on the size of the last contiguous block by letting it start at $k+1$. Then we need to compute the minimum imbalance by partitioning the first k numbers into $j-1$ blocks. But this is exactly $B[k, j-1]$! Now the minimum configuration is achieved by taking the minimum imbalance in all these cases, which is what the algorithm does, proving the claim. The "best" partition can then be returned by returning $B[n, k][1]$.

We can in fact improve this algorithm by deleting column i of $B[i, j]$ once we are done using it, so we can reduce the space complexity for the lists from $n^2 O(k)$ to $O(nk)$.

Finally, we examine the time and space complexity of the algorithm. For time complexity, the outer loop has $O(k)$ operations, and each of which takes $O(n^2)$ operations, so the time complexity is $O(n^2 k)$. For space complexity, we take $O(n^2)$ initializing the $a_{i,j}$'s, the first elements of the $B[i, j]$'s take $O(n)$ in total (using the deletion method), and the second element of the $B[i, j]$'s can take $O(nk)$ space as discussed above. Now note that we can assume WLOG that $k \leq n$ (else we can just leave some partitions empty), so the space complexity becomes $O(n^2 + kn) = O(n^2)$.

(b) **(5 points)** Explain how your algorithm would change if the imbalance was redefined to be

$$\sum_i \left| w(i) - \left(\sum_{\ell=1}^{n} A[\ell] \right) / (k+1) \right|.$$

In this case, we instead take redefine temp into the sum of $B[k, j-1][0]$ and $|a_{k+1,i} - c|$. The necessary arguments and results are nearly identical. This works because the method wold then sum through the required absolute value differences instead of taking the maximum, which is what we want. The time and space complexities are the same.

2. (a) **(20 points)** Suppose we want to print a paragraph neatly on a page. The paragraph consists of words of length $\ell_1, \ell_2, \ldots, \ell_n$. The maximum line length is M. (Assume $\ell_i \leq M$ always.) We define a measure of neatness as follows. The extra space on a line (using one space between words) containing words ℓ_i through ℓ_j is $M - j + i - \sum_{k=i}^{j} \ell_k$. The penalty is the sum over all lines **except the last** of the **cube** of the extra space at the end of the line. This has been proven to be an effective heuristic for neatness in practice. Find a dynamic programming algorithm to determine the neatest way to print a paragraph. Of course you should provide a recursive definition of the value of the optimal solution that motivates your algorithm.

   (Refer to next part for some of the variable names.) We first make some simplifications for this problem. Note that the definition of a penalty is equivalent to $(M + 1) - \sum_{k=i}^{j}(l_k + 1)$, so we can "pretend" that each word ends in a space and that the maximum characters per line is $M + 1$ and we wish to minimize the sum of the cubes of the new penalty $M - \sum_{k=i}^{j} l_k$.

   In order to do this, we use dynamic programming by storing the minimum penalties and the line broken text when we are using only the first i words (for each $0 \leq i \leq n$, where n is the number of words), where we are also counting the penalty of the last line (code at the end of this proof–maxLength here is M in the problem).

   In fact, we claim that in the algorithm below, the resVals array contains the minimum penalties for each i (starting with 0 for the empty word) and that the resText array contains the optimal line breaks. We prove this by induction on i. The base case $i = 0$ is clear. Now suppose that we have proven this for all $j \leq i$, and we will prove this for $i + 1$. Now note that when we add the next word, all the possibilities are obtained by determining the position of the word the last line starts (where it ends at word $i + 1$). This is achieved by the for loop using j as an iterator. We then check the length of the last line based on this (has to be at most M). The total penalty here is then the total penalty of the previous $j - 1$ words (optimally broken by induction), plus the penalty of the last line (as long as it is valid), which is done in the if statement. The string that captures this optimal position of breaks is then recorded together with the minimum penalty found, in the $i + 1^{\text{th}}$ position of the res arrays. Then by definition, since we are taking the minimum of all possibilities, it follows that for $i + 1$ this minimum is also the optimal. This completes the inductive step and proves the claim.

   Finally, to remove the last line penalty, we again use casework on the starting word index of the last line (expressed by i in the last for loop). We record the minimum penalty of the strings and the string that achieves this for as long as the last line is valid (has length sumLengths[n]-sumLengths[i], since this is the number of characters between the word i and the last word, and we just need this to be at most M) and output the result. By definition then, this would be the desired output.

   This algorithm runs in $O(n^2)$ time: $O(n)$ for the first for loop, $O(n^2)$ in the second for loop, and $O(n)$ in the third. This takes $O(n)$ space by initializing the arrays and other variables.

   (b) **(20 points)** Determine the minimal penalty, and corresponding optimal division of words into lines, for the following review of the Season 1 Buffy DVD, apparently written by Ryan Crackell for the Apollo Guide, for the cases where $M = 40$ and $M = 72$.

   (You can find the answer by whatever method you like, but we recommend coding your algorithm from the previous part. You don't need to submit code.)

   (The text of the review may be easier to copy-paste from the tex source than from the pdf. If you copy-paste from the pdf, check that all the characters show up correctly—the "ff" in "Buffy", among others, often doesn't.)

   Buffy the Vampire Slayer fans are sure to get their fix with the DVD release of the show's first season. The three-disc collection includes all 12 episodes as well as many extras. There is a collection of interviews by the show's creator Joss Whedon in which he explains his inspiration for the show as well as comments on the various cast members. Much of the same material is covered in more depth with Whedon's commentary track for the show's first two episodes that make up the Buffy the Vampire Slayer pilot. The most interesting points of Whedon's commentary come from his explanation of the learning curve he encountered shifting from blockbuster films like Toy Story to a much lower-budget television series. The first disc also includes a short interview with David Boreanaz who plays the role of Angel. Other features include the script for the pilot episodes, a trailer, a large photo gallery of publicity shots and in-depth biographies of Whedon and several of the show's stars, including Sarah Michelle Gellar, Alyson Hannigan and Nicholas Brendon.

The output of the code can be found under the code below.

```
import java.util.*;
import java.io.*;

public class LineBreaker {
```

```
public static int cube(int n) {
    return n*n*n;
}

public static void breaker(String text, int maxLength) {
    String[] words = text.split(" ");
    int n = words.length;

    // initialize stuff for dp bookkeeping
    int[] sumLengths = new int[n+1];
    int[] wordLengths = new int[n];

    // add in 0 and "" for base case
    int[] resVals = new int[n+1];
    String[] resText = new String[n+1];

    resVals[0] = 0;
    resText[0] = "";

    // account for "word 0" for later convenience
    sumLengths[0] = 0;
    int tempSum = 0;
    for (int i = 0; i < n; i++) {
        int temp = words[i].length() +1;
        tempSum += temp;
        wordLengths[i] = temp;
        sumLengths[i+1] = tempSum;
    }

    // from arguments earlier
    maxLength += 1;
    // set min penalty to impossibly large number for replacement
    int minPenalty = cube(maxLength) +1;

    // setup for dp
    // find smallest penalty for first i words
    // penalty includes last line penalty
    for (int i = 0; i < n; i++) {
        // we will set this to be the "inducted" value
        int minTempVal = minPenalty;
        String tempStr = "";
        for (int j = 0; j <= i; j++) {
            // what happens when we start last line at j?
            int lastLine = sumLengths[i+1] - sumLengths[j];
            int temp = resVals[j] + cube(maxLength - lastLine);

            if (lastLine <= maxLength && temp < minTempVal) {
                minTempVal = temp;
                tempStr = resText[j] + '\n' +
                    text.substring(sumLengths[j], sumLengths[i+1] -1);
            }
        }
        resVals[i+1] = minTempVal;
        resText[i+1] = tempStr;
    }

    // System.out.println(Arrays.toString(resVals));
```

```
        // now, we bash for the best last line
        // we already have dp array so we gucci
        int bestVal = minPenalty;
        String res = "";
        for (int i = 0; i < n; i++) {
            // if can in last line
            if (sumLengths[n] - sumLengths[i] < maxLength && resVals[i] < bestVal) {
                bestVal = resVals[i];
                res = resText[i] + '\n' + text.substring(sumLengths[i]);
            }
        }

        System.out.printf("penalty: %d\n", bestVal);
        System.out.println(res);
    }

    public static void main(String[] args) {
        String text = "Buffy the Vampire Slayer fans are sure to get their fix with the
        breaker(text, 40);
        breaker(text, 72);
    }
}
```

OUTPUT:

```
breaker(text, 40);

penalty: 2183

Buffy the Vampire Slayer fans are
sure to get their fix with the DVD
release of the show's first season.
The three-disc collection includes all
12 episodes as well as many extras.
There is a collection of interviews
by the show's creator Joss Whedon in
which he explains his inspiration for
the show as well as comments on the
various cast members.  Much of the
same material is covered in more depth
with Whedon's commentary track for the
show's first two episodes that make
up the Buffy the Vampire Slayer pilot.
The most interesting points of Whedon's
commentary come from his explanation
of the learning curve he encountered
shifting from blockbuster films like Toy
Story to a much lower-budget television
series. The first disc also includes
a short interview with David Boreanaz
who plays the role of Angel. Other
features include the script for the
pilot episodes, a trailer, a large photo
gallery of publicity shots and in-depth
biographies of Whedon and several of the
```

show’s stars, including Sarah Michelle Gellar, Alyson Hannigan and Nicholas Brendon.

breaker(text, 72);

penalty: 2104

Buffy the Vampire Slayer fans are sure to get their fix with the DVD release of the show’s first season. The three-disc collection includes all 12 episodes as well as many extras. There is a collection of interviews by the show’s creator Joss Whedon in which he explains his inspiration for the show as well as comments on the various cast members. Much of the same material is covered in more depth with Whedon’s commentary track for the show’s first two episodes that make up the Buffy the Vampire Slayer pilot. The most interesting points of Whedon’s commentary come from his explanation of the learning curve he encountered shifting from blockbuster films like Toy Story to a much lower-budget television series. The first disc also includes a short interview with David Boreanaz who plays the role of Angel. Other features include the script for the pilot episodes, a trailer, a large photo gallery of publicity shots and in-depth biographies of Whedon and several of the show’s stars, including Sarah Michelle Gellar, Alyson Hannigan and Nicholas Brendon.

3. **(25 points)** Consider the following string compression problem. We are given a dictionary of m strings, each of length at most k. We want to encode a data string of length n using as few dictionary strings as possible. For example, if the data string is bababbaababa and the dictionary is (a,ba,abab,b), the best encoding is (b,abab,ba,abab,a). Give an $O(nmk)$ algorithm to find the length of the best encoding or return an appropriate answer if no encoding exists.

   Let $T = \{t_1, t_2, \ldots t_m\}$ be the set of strings (dictionary) and s the string we wish to compress. Then from the problem we know that `len(s) = n`. We define the *score* of a string to be the minimum number of dictionary words in T we need to concatenate to obtain the string. We then perform a DP algorithm as follows: construct an array dp of length $n + 1$. Our goal is that for each $k \leq n$, $dp[k]$ will contain the score of the string formed by taking the first k letters of s, together with the order of concatenations of the t_i's. We do this recursively. First initialize $dp[0] = 0$ and its concatenation list to be the empty list []. Then for $0 \leq k \leq n$, consider the string formed by the first k letters of s. Then for each $t \in T$, we check whether or not t can be placed in the last position of the substring (so their last letters match, down until the first letter of t). If we cannot, we move on to the next element of T. If we can, we set $dp[k] = dp[k - len(t)] + 1$ and set the concatenation list for $dp[k]$ to be the concatenation list of $dp[k - len(t)]$, appended by t if the score $dp[k]$ is uninitialized or becomes lower if we replace it. If at some k we cannot initialize $dp[k]$, we set it equal to ∞. Finally, after the loop, we return $dp[n]$. This returns an integer which is the score, or the length of the best encoding, and a list, which is how we would achieve the best encoding.[1] If the score is ∞, this means the string cannot be compressed through the dictionary T.

   (Here a substring means a contiguous block of characters starting from the first) We prove correctness by induction. The claim is clear for $k = 0$. Now suppose it is true for all $0 \leq l \leq k - 1$ where $k \geq 1$. Then note that when the algorithm loops over the m entries of T, if none fit in the tail (last few letters) of the current substring then this substring cannot be constructed from T, so we cannot use this substring to "tower down" in future substrings. Now for the elements $t \in T$ that do fit in the tail of the substring, by the inductive hypothesis setting $dp[k]$ would set it to 1 plus the minimum score of every shorter substring that can be constructed, which therefore means this is the best compression for this substring, completing the inductive hypothesis.[2] The concatenation lists are correct because the concatenation list must be the newly added string $t \in T$ at the end, before which must be the concatenation list of the previous substring.[3] Therefore, $dp[n]$ does indeed contain the best score, which proves correctness.

   For the run time, note that checking if each dictionary word can be placed in the last position takes $O(k)$ time, since we need to check each character and we have to do this at most k times (max length of the dictionary strings). Replacing the values in dp takes $O(1)$ time. Meanwhile, we have to loop this over the m strings in T, and do this for the n letters of s, so the time complexity becomes $O(nmk)$, as desired. The space complexity is $O(n)$ from the dp array.

[1] We technically don't need the lists, in that case we can ignore everything we talked about regarding the concatenation lists. Although it would be pretty useless to determine only the best length and not how to do it...

[2] This is because any concatenation of strings from T would always need to fit some $t \in T$ in the tail first, then fitting in the other t's in the remaining letters. So what we did exhausts all cases.

[3] Again this isn't needed for this problem but is needed if we want to know what the concatenations are, so feel free to ignore this as well.

4. **(0 points, optional)**[4] Consider an algorithm for integer multiplication of two n-digit numbers where each number is split into three parts, each with $n/3$ digits.

   (a) Design and explain such an algorithm, similar to the integer multiplication algorithm presented in class. Your algorithm should describe how to multiply the two integers using only six multiplications on the smaller parts (instead of the straightforward nine).

   (b) Determine the asymptotic running time of your algorithm. Would you rather split it into two parts or three parts?

   (c) Suppose you could use only five multiplications instead of six. Then determine the asymptotic running time of such an algorithm. In this case, would you rather split it into two parts or three parts?

   (d) (Harder; the instructors will be impressed if you solve it.) Find a way to use only five multiplications on the smaller parts. Can you generalize to when the two initial n-digit numbers are split into k parts, each with n/k digits? Hint: also consider multiplication by a constant, such as 2; note that multiplying by 2 does not count as one of the five multiplications. You may need to use some linear algebra.

[4] We won't use this question for grades. Try it if you're interested. It may be used for recommendations/TF hiring.