CS 124 Homework 4: Spring 2022

Your name: Jamin Liu

Collaborators: Elizabeth Li

No. of late days used on previous psets: 6
No. of late days used after including this pset: 8

Homework is due Wednesday at midnight 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 assignme nt must be a nonnegative integer at most **two** (college)/**four** (extension school).

Try to make your answers as clear and concise as possible; style may count in your grades. 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.

You can collaborate with other students that are currently enrolled in this course in brainstorming and thinking through approaches to solutions but you should write the solutions on your own: you must wait one hour after any collaboration or use of notes from collaboration before any writing in your own solutions that you will submit.

For all homework problems where you are asked to give an algorithm, you must prove the correctness of your algorithm and establish 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. Consider an algorithm for integer multiplication of two n-digit numbers where each number is split into three parts, each with $n/3$ digits. (You may assume that n is a power of 3.)

   (a) **(15 points)** 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).

   Solution:

   To describe an algorithm for integer multiplication of two n-digit numbers where each number is split into three parts, each with $n/3$ digits, consider the following. Let I be the first n-digit number and J be the second n-digit number. Let a be the first $\frac{n}{3}$ digits of I, let b be the next $\frac{n}{3}$ digits, and let c be the last $\frac{n}{3}$ digits. Similarly, let d be the first $\frac{n}{3}$ digits of J, let e be the next $\frac{n}{3}$ digits, and let f be the last $\frac{n}{3}$ digits. Note that I and J can be written in the following manner:

$$I = 10^{2n/3} \cdot a + 10^{n/3} \cdot b + c$$
$$J = 10^{2n/3} \cdot d + 10^{n/3} \cdot e + f$$

As such, we may write the product of I and J as the following:

$$\begin{aligned} I \cdot J &= (10^{2n/3} \cdot a + 10^{n/3} \cdot b + c) \cdot (10^{2n/3} \cdot d + 10^{n/3} \cdot e + f) \\ &= 10^{4n/3} \cdot ad + 10^{n} \cdot (ae + bd) + 10^{2n/3} \cdot (af + be + cd) + 10^{n/3} \cdot (bf + ce) + cf \end{aligned}$$

We seek to derive the expressions $ad, (ae + bd), (af + be + cd), (bf + ce)$, and cf using only six multiplication operations. To do this, consider the following six multiplication operations:

$$\begin{gathered} a \cdot d = ad \\ b \cdot e = be \\ c \cdot f = cf \\ (a+b) \cdot (d+e) = ad + ae + bd + be \\ (b+c) \cdot (e+f) = be + bf + ce + cf \\ (a+c) \cdot (d+f) = ad + af + cd + cf \end{gathered}$$

Thus, using these, we are able to derive our desired expressions as follows:

$$\begin{aligned} ad &= ad \\ ae + bd &= (a+b)(d+e) - ad - be \\ af + be + cd &= (a+c)(d+f) - ad - cf + be \\ bf + ce &= (b+c)(e+f) - be - cf \\ cf &= cf \end{aligned}$$

Accordingly, consider the following algorithm for integer multiplication of two n-digit numbers where each number is split into three parts, each with $n/3$ digits:

1. Let I be the first n-digit number and J be the second n-digit number. Let a be the first $\frac{n}{3}$ digits of I, let b be the next $\frac{n}{3}$ digits, and let c be the last $\frac{n}{3}$ digits. Similarly, let d be the first $\frac{n}{3}$ digits of J, let e be the next $\frac{n}{3}$ digits, and let f be the last $\frac{n}{3}$ digits.

2. Complete the following six multiplication operations: $ad, be, cf, (a+b)(d+e), (b+c)(e+f)$, and $(a+c)(d+f)$. 3. Add, subtract, and scale by powers of 10 according to

the following formula to give the product of I and J:

$$\begin{aligned} I \cdot J = &10^{4n/3} \cdot ad \\ &+ 10^{n} \cdot ((a+b)(d+e) - ad - be) \\ &+ 10^{2n/3} \cdot ((a+c)(d+f) - ad - cf + be) \\ &+ 10^{n/3} \cdot ((b+c)(e+f) - be - cf) \\ &+ cf \end{aligned}$$

Proof of Correctness:

Intuitively, following the problem statement, we may express I and J as the sum of of a, b, c and d, e, f respectively and multiplied by corresponding powers of 10. Next, by simple algebra as shown above, we know that the expression $I \cdot J$ is equivalent to the expression given in part 3 of our algorithm. Moreover, our expression for $I \cdot J$ only takes six multiplications of numbers on the order of $\frac{n}{3}$ digits instead of the straightforward nine. Thus, our algorithm is correct.

(b) **(7 points)** Determine the asymptotic running time of your algorithm. Would you rather split it into two parts or three parts?

Solution:

Let $T(n)$ be the time it takes to multiply two n-digit numbers. The the recurrence for $T(n)$ for our algorithm is:

$$T(n) = 6T(n/3) + O(n)$$

The $6T(n/3)$ term comes from the six multiplication operations of $n/3$ digit numbers from part (a) and the $O(n)$ term is the time to combine these expressions into the final product using addition, subtraction, and shifts. Thus, by the Master Theorem, the asymptotic running time of our algorithm is $O(n^{log_3 6}) = O(n^{1.63})$.

If we were to split our n-digit number into two parts, we would then be following Karatsuba's algorithm, which we know from lecture has asymptotic running time of $O(n^{1.59})$. Since this is a faster asymptotic running time than our algorithm proposed in part (a), we would rather split our n-digit number into two parts.

(c) **(7 points)** 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?

Solution:

If we were somehow able to use only five multiplications instead of six, the recurrence for $T(n)$ for our algorithm would be:

$$T(n) = 6T(n/3) + O(n)$$

Thus, by the Master Theorem, the asymptotic running time of our algorithm would be $O(n^{log_3 5}) = O(n^{1.47})$. As discussed above, the asymptotic running time of Karatsuba's algorithm, which splits our n-digit number into two parts, is $O(n^{1.59})$. Thus, since the proposed algorithm which splits our n-digit number into three parts and uses five multiplications has a faster asymptotic running time than Karatsuba's, we would rather split our n-digit number into three parts.

(d) **(0 points, optional)**[1] Find a way to use only five multiplications on the smaller parts.

(e) **(0 points, optional)**[2] (Harder; Adam will be impressed if you solve it.) Generalize the previous subproblem to when the two initial n-digit numbers are split into k parts, each with n/k digits. (You may assume that n is a power of k.) 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.

2. (a) **(25 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.

Solution

Assuming our paragraph contains a nontrivial number of words, consider the following algorithm for determining the neatest way to print a paragraph. Note that the neatest way to print the paragraph is the way which minimizes the penalty.

Definition:

Let the n words within our paragraph be $w_1, w_2, \ldots, w_n$ with each w_i of length ℓ_i. Let L be a list of length n where each $L[i]$ stores the minimized penalty (neatest) way of arranging words w_1 through w_i. Let F store the index of the first word of the last line of the arrangement of words w_1 through w_i with minimized penalty. We calculate all values of L and F recursively till we attain our minimized penalty $L[n]$. We then use the information stored within F to construct our paragraph upon completing our recursion.

Initialization:

To aid us in this problem, we initialize the following lists and matrices:

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

Construct a list L of length n, where each $L[i]$ for $i \in \{1, 2, \ldots, n\}$ stores the minimal penalty value for arranging words w_1 through w_i with all values of L initialized to ∞.

Construct a list F of length n, where each $F[i]$ for $i \in \{1, 2, \ldots, n\}$ stores the index of the first word on the last line of the neatest arrangement of words w_1 through w_i.

Construct an $n \times n$ matrix E where each entry $E[i][j]$ stores the amount of extra space on a line containing words w_i through w_j. Using the formula given in the problem statement, we calculate $E[i, j]$ as follows

$$E[i, j] = M - j + i - \sum_{k=1}^{j} \ell_k.$$

Note that $E[i, j]$ is only well defined when $i \leq j$ as the words adhere to a specific ordering, meaning we cannot have a line containing words w_i through w_j, if the word w_j comes before w_j. As such, all entries of E below the diagonal are NULL.

Construct an $n \times n$ matrix E where each entry $E[i][j]$ stores the cube of extra space at the end of a line containing words w_i through w_j. In the case that $j = n$, or that we are on the last line of the paragraph, $C[i, j] = 0$. Moreover, if $E[i, j] < 0$, meaning the line containing words w_i through w_j has a length greater than M or that words w_i through w_j do not fit on a single line, set $C[i, j] = \infty$ so that our algorithm will never include these words on a single line (since we are trying to minimize our penalty). In all other cases let $C[i, j] = (E[i, j])^3$. Similar to E, note that $C[i, j]$ is only well defined when $i \leq j$ as the words adhere to a specific ordering, meaning we cannot have a line containing words w_i through w_j, if the word w_j comes before w_j. As such, all entries of C below the diagonal are NULL.

Recurrence:

To describe our recurrence constructing L, we start first with our base case. In the case when $i = 1$, we calculate the penalty of arranging the first word w_1 on a single line. Since we know that all $\ell_i < M$, we know that w_1 fits on a single line and is the only word on said line, so $L[1] = C[1, 1]$. For the first word of our last line, since we only have a single word on a single line, we have $F[1] = 1$.

Proceeding, our recurrence for constructing L follows. Note that for all $i \in \{2, 3, \ldots, n\}$, for any neatest arrangements of words w_1 through w_i, there exists a series of words on the last line, let us call them words w_k through w_i, such that $1 \leq k \leq i$. Since we recursively construct proceeding values of L, all lines preceding this last line must be in the neatest arrangement such that we have the minimized penalty $L[k-1]$, giving the minimized penalty of all preceding words words w_1 through w_{k-1}. Next, let us note that there are multiple possible arrangements for the line break between the second-to-last and last line, as k could be any value between 1 and i. As such, to find the optimal arrangement, we wish to iterate through each of these possible line break arrangements to determine which of these gives the minimized penalty. Let us also not that there are a maximum of $\frac{M}{2}$ words per line, since each word has a space following it. As such the recurrence for constructing L is given by:

$$L[i] = \min(L[k-1] + C[k,i]) \text{for all } k \text{ such that } 1 \leq k \leq \min(i, \frac{M}{2}).$$

For the k chosen which minimizes the value of $L[k-1] + C[k,i]$, we set the index of the first word on the last line $F[i] = k$. For $L[i]$, we have stored the minimized penalty for formatting words w_1 through w_i. Once we have finished our recursion on all n words, we can construct our neatest paragraph with minimized penalty $L[n]$ by tracing our line breaks through F. Our optimal arrangement's last line will contain words from the indices $F[n]$ to n, our optimal arrangement's second-to-last line will contain words from the indices $F[F[n]-1]$ to $F[n]-1$, continuing till we have constructed all line breaks within the paragraph and have reached $F[1]$.

Space:

The space complexity of the matrices and lists described above are as follows:

L is a list of length n requiring $O(n)$ space.
F is of length n requiring $O(n)$ space.
E is an $n \times n$ matrix requiring $O(n^2)$ space.
C is an $n \times n$ matrix requiring $O(n^2)$ space.

As such, the overall space complexity of our algorithm is $O(n^2)$.

Runtime

To analyze the runtime complexity of our algorithm, we will analyze the runtime it takes to compute a unit of space for each data structure described above, then multiply by the corresponding space complexity to generate the overall runtime.

Computing each entry of the list L takes $O(n)$ time, as we require n constant time comparisons to identify the optimal $L[i]$ assignment. Since L is built recursively, note that we are able to access previous entries of L in $O(1)$ as they have already been calculated. Thus, constructing L takes $O(n)$ time and accessing the minimized penalty $L[i]$ takes $O(1)$ time.

Computing each entry of the list F takes $O(1)$ time, as we simply identify the index of the first word of the last line. Thus, constructing F takes $O(n)$ time.

Once F is constructed, reiterating through F to access line break assignments to generate our desired paragraph takes $O(n)$ time.

Assigning each of the $O(n^2)$ entries in E takes $O(1)$ time making use of a function for computing $E[i,j]$ modified from the definition of the extra space function:

$$E[i,j] = \begin{cases} E[i,j-1] - 1 - \ell_j & \text{if } i < j \\ M - \ell_i & \text{if } i = j \end{cases}$$

As discussed above, $E[i,j]$ is only well-defined for $i \leq j$, so this function's assumption of $j-1$ is acceptable, meaning computing each $E[i][j]$ takes $O(1)$. As such, the runtime of constructing E is $O(n^2)$.

Assigning each of the $O(n^2)$ entries in C takes $O(1)$ time as we either set $C[i,j]$ to 0, ∞, or $(E[i,j])^3$, so each $C[i][j]$ requires a comparison operations and an exponent operation on $E[i,j]$. As such, the runtime of constructing C is $O(n^2)$.

Thus, the overall runtime of our algorithm is $O(n^2)$.

Proof of Correctness:

To prove the correctness of our algorithm, we will justify that our recurrence gives the neatest arrangement of words, or the arrangement of words which minimizes our penalty.

First, for our base case where $i = 1$, $L[1] = C[1,1]$ and $F[1] = 1$ while all other values of L are set to ∞. Next, for all proceeding i, we consider all possible k where $1 \leq h \leq \min(i, \frac{M}{2})$, guaranteeing that we have exhausted all possible positions for a line break for each i. As previously mentioned, since L is built through our recursion, we know we have correctly calculated all minimized penalties preceding $L[i]$. Using $L[k-1] + C[k,i]$ with k such that penalty is minimized for all $L[i]$, we are guaranteed to have the minimized penalty for $L[n]$ and thus, the correct line breaks recorded within F. Finally, by reiterating through F as per our description above, we are ensured that our algorithm constructs the paragraph with the optimal line breaks in order to minimize penalty. As such, our algorithm correctly determines the neatest way to print the paragraph, so it is correct.

(b) **(20 points)** Determine the minimal penalty, and corresponding optimal division of words into lines, for the text of this question, from the first 'Determine' through the last 'correctly.)', for the cases where M is 40 and M is 72. Words are divided only by spaces (and, in the pdf version, linebreaks) and each character that isn't a space (or a linebreak) counts toward the length of a word, so the last word of the question has length eleven, counting the period and the right parenthesis. (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.)

Solution:

For $M = 40$, the paragraph is formatted in the following manner:

Determine the minimal penalty, and
corresponding optimal division of
words into lines, for the text of this
question, from the first ‘Determine’
through the last ‘correctly.)’, for
the cases where M is 40 and M is 72.
Words are divided only by spaces (and,
in the pdf version, linebreaks) and
each character that isn’t a space (or a
linebreak) counts toward the length of
a word, so the last word of the question
has length eleven, counting the period
and the right parenthesis. (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 penalty incurred according to our algorithm for when $M = 40$ is 1116.

For $M = 72$, the paragraph is formatted in the following manner:

Determine the minimal penalty, and corresponding optimal division
of words into lines, for the text of this question, from the first
‘Determine’ through the last ‘correctly.)’, for the cases where M is 40
and M is 72. Words are divided only by spaces (and, in the pdf version,
linebreaks) and each character that isn’t a space (or a linebreak)
counts toward the length of a word, so the last word of the question
has length eleven, counting the period and the right parenthesis. (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 penalty incurred according to our algorithm for when $M = 72$ is 885.

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.

   Solution:

   Consider the following $O(nmk)$ algorithm for finding the length of the best encoding:

   Definition:

   Let s be our data string of length n. For this data string, let $s[i]$ be the ith character of the string and let $s[i : j]$ be the substring of s from the ith character including the jth character. Let L be a list of length n where $L[i]$ stores the length of the best encoding for each substring $s[1 : i]$ for $i \in \{1, 2, \ldots, n\}$. By recursively calculating each $L[i]$, we reach $L[n]$ which gives the length of the best encoding for s. Moreover, we initialize all values of L to ∞ and only update the value of a given $L[i]$ if a valid encoding exists for $s[1 : i]$. Thus, if no valid encoding exists for s, we will be able to deduce this and indicate that no valid coding exists.

   Initialization:

   Construct a list L of length n with all values initialized to ∞.

   Recurrence:

   To describe our recurrence for constructing L, we start first with our base case. In the case that $i = 1$, we iterate through all m strings of the dictionary to determine whether one matches $s[1]$. If we find such a string, we set $L[1]$ to 1. If not, we do not alter $L[1]$, so $L[1]$ remains ∞ as initialized.

   Proceeding, our recurrence for constructing L follows. To determine $L[i]$, first we consider the case in which there exists a valid encoding of the substring $s[1 : i]$. In this case, there must exist a string $s[j : i]$ in the dictionary for a j such that $\max\{1, i + 1 - k\} \leq j \leq i$ (k is the maximum length of strings in the given dictionary). As such, the best encoding for $s[1 : i]$ is composed of the substrings $s[1 : j]$ and $s[j : i]$. Thus, the length of the best encoding for $s[1 : i]$ is then given by $L[j - 1] + 1$, or the length of the best encoding for the first substring $s[1 : j]$ plus the additional string $s[j : i]$ used from the dictionary. Since there could exist multiple $s[j : i]$ that create a valid encoding, we take the minimum of all possible $L[j - 1]$ values, for all j such that $\max\{1, i + 1 - k\} \leq j \leq i$. In the case that there exists no valid encoding for $s[1 : i]$, meaning no string in the dictionary ends with $s[i]$, we do not alter $L[i]$, so $L[i]$ remains ∞ as initialized. As such the recurrence for constructing L is given by:

$$L[i] = \begin{cases} \infty & \text{if there exists no valid encoding of } s[1:i] \\ \min\{L[j-1]\} + 1 & \text{if there exists a valid encoding of } s[1:i] \\ & j \text{ such that } max\{1, \text{i-k+1}\} \leq j \leq i \end{cases}$$

Once we have finished our recursion on all n letters and have computed $L[i]$ for $i \in \{1, 2, \ldots, n\}$, we return $L[n]$, the length of the best encoding for s. In the case that we have $L[n] = \infty$, we return that no valid encoding exists.

Space:

L is a list of length n requiring $O(n)$ space. This is our only data structure, so the overall space complexity of our algorithm is $O(n)$.

Runtime:

To analyze the runtime complexity of our algorithm, we will analyze the runtime it takes to compute a unit of space for each data structure described above, then multiply by the corresponding space complexity to generate the overall runtime.

First note that initializing the entries of L to ∞ takes $O(n)$ time. To calculate $L[i]$, we search the dictionary of length m for a string which matches substring $s[j:i]$, and we do this for all k substrings where we have j such that $max\{1, i-k+1\} \leq j \leq i$. Thus, for every $L[i]$ we must make $m \cdot k$ comparisons. Since L has n entries in total and we must calculate all entries to determine the length of the best encoding, the overall runtime of computing the entries of L is $O(mnk)$. As such, the overall runtime of our algorithm is $O(mnk)$.

Proof of Correctness:

To prove the correctness of our algorithm, we will justify that our recurrence gives the length of the best encoding or indicates if no such encoding exists.

First for our base case of the substring $s[i]$, by iterating through the dictionary of strings to determine whether any match this first character, we guarantee that we are not ignoring the case in which the best encoding includes a single-charactered string. For the following steps of our recurrence, to calculate each $L[i]$, we check the dictionary for a string which matches substring $s[j:i]$, and we do this for all k substrings where we have j such that $max\{1, i-k+1\} \leq j \leq i$. Doing this guarantees that we have searched all possible valid encodings for the substring $s[1:i]$. Moreover, in the case of multiple valid encodings, taking the minimum using the expression $\min\{L[j-1]\} + 1$ ensures that we use the length of the best encoding.

Moreover, since we calculate values of L recursively with increasing values of i, we know that we have correctly calculated all previous $L[1], L[2], \ldots, L[i-1]$ when computing $L[i]$, meaning that the expression $\min\{L[i'-1]\}+1$ will return us an accurate length for the best encoding.

In the case that there exists no valid encoding, our function will not have updated the value of $L[n]$ from the value it was initialized as, meaning that it remains ∞. This then allows us to deduce that no valid encoding exists.

As such, our algorithm is correct.

4. **(25 points)** Given a positive integer n and, for each pair (i,j) with $0 \le i \le j < n$, an integer $X_{i,j} \in \{0,1\}$, give an algorithm to find integers $C_{i,j} \in \{0,1\}$ for each pair (i,j) with $0 \le i \le j < n$ such that:

   (a) For all i, j, and k with $i \le j \le k$, if $C_{i,j} = 1$ and $C_{j,k} = 1$, then $C_{i,k} = 1$.

   (b) For all i, j, and k with $i \le j \le k$, if $C_{i,k} = 1$, then $C_{i,j} = 1$ and $C_{j,k} = 1$.

   (c) Subject to the above,

   $$2\sum_{X_{ij}=1} C_{ij} - \sum_{X_{ij}=0} C_{ij}$$

   is as large as possible.

   Nothing below is necessary to understand the problem, but is included as a description of why you might care:

   In biology and biophysics, scientists are sometimes interested in *topologically associating domains* (TADs), or a contiguous region of DNA that interacts with itself in some way. TAD architectures vary by cell type: lung cells may have different TADs than red blood cells. One question one may ask is whether there are certain TADs that are conserved across cells, which may indicate some fundamental biological role of these TADs (as opposed to lung-specific role). Formally, let X_{ij} represent the number of cell types for which $DNA[i]$ to $DNA[j]$ is contained within in a single TAD. Let T be the number of cell types that we've sequenced, so $0 \le X_{ij} \le T$ for all i, j. $\mathbf{X} = \{X_{ij} | 0 \le i < j \le n\}$ We are interested in building a consensus TAD map, such that $C_{ij} \in \{0,1\}$.

   One method of building a consensus TAD requires maximizing a function that reduces to:

   $$argmax_{\mathbf{C}} \sum_{T \ge X_{ij} > 0} 2C_{ij} - \sum_{X_{ij}=0} C_{ij}$$

   There are a few constraints. As we noted, TADs are contiguous, so if $C_{ij} = 1$ and $C_{jk} = 1$ then $C_{ik} = 1$. Similarly, if $C_{ik} = 1$ then for all j with $i < j < k$, $C_{ij} = 1$ and $C_{jk} = 1$. Intuitively, the first term of the sum favors long TADs, as setting $C_{ij} = 1$ for a very long region would increase this sum. However, it would be penalized by the second term, because for long enough regions, even if $X_{ij} = 2$, there will likely be subregions for which $X_{k\ell} = 0$ for $i < k < \ell < j$.

Solution:

First, let us develop some intuition for the problem. For a positive integer n and $X_{i,j} \in \{0,1\}$ for all $0 \leq i \leq j < n$, we wish to assign values of $C_{i,j} \in \{0,1\}$ for all $0 \leq i \leq j < n$ such that we maximizes the value of $2\sum_{X_{i,j}=1} C_{i,j} - \sum_{X_{i,j}=0} C_{i,j}$ while adhering to the constraints described in (a) and (b). As such, intuitively, our goal is to preserve the assignment of an $X_{i,j}$ in its corresponding $C_{i,j}$ for as many $X_{i,j}$ as possible. Thus, for any $X_{i,j} = 1$, we wish to assign its corresponding $C_{i,j}$ to be 1 as well, and similarly, for any $X_{i,j} = 0$, we wish to assign its corresponding $C_{i,j}$ to 0. Given the score function $2\sum_{X_{i,j}=1} C_{i,j} - \sum_{X_{i,j}=0} C_{i,j}$, we observe that the term $\sum_{X_{i,j}=1} C_{i,j}$ is multiplied by a factor of 2. We may take this to mean that we prioritize setting a $C_{i,j}$ to 1 for a $X_{i,j} = 1$ over setting a $C_{i,j}$ to 0 for a $X_{i,j} = 0$.

To help us visualize this problem, let us imagine an $n \times n$ matrix M storing all $X_{i,j}$ such that $M[i][j] = X_{i,j}$. Since we only have X_{ij} for $i \geq j$, values below the diagonal do not exist. Also note that when assigning $C_{i,j}$ values to 1, we always do so in right-isosceles-triangular configurations along the diagonal due to our constraints (a) and (b). Together, our constraints dictate that $C_{i,k} = 1 \iff C_{i,j} = 1, C_{j,k} = 1$ for $i \leq j \leq k$. As such, for any given $C_{i,k} = 1$, all cells to the left and below must also be 1, creating a right-isosceles-triangle configuration where the hypotenuse is along the diagonal of the matrix.

Definition:

Let L be a list of length n, where $L[i]$ gives the maximum value of the score function $2\sum_{X_{i,j}=1} C_{i,j} - \sum_{X_{i,j}=0} C_{i,j}$ while assuming $n = i + 1$, given a positive integer n and an $X_{i,j} \in \{0,1\}$ for each (i,j) pair with $0 \leq i \leq j < n$. Let $A[i]$ describe the assignment of $C_{i,j}$ values to set to 1 for an $i \times i$ matrix in order achieve the maximum score function value $L[i]$. Using $A[n-1]$, assign all $C_{i,j}$ in the matrix C such that we have the maximum score function value $L[n-1]$ and return the matrix C.

Initialization:

To aid us in this problem, we initialize the following lists and matrices:

Store all $X_{i,j}$ in an $n \times n$ matrix M. Each row represents i values from 0 to $n-1$, and each column represents j values from 0 to $n-1$ such that $M[i][j] = X_{i,j}$. Moreover, note that since we only have X_{ij} for $i \geq j$, all values of M below the diagonal are NULL.

Given a set of $X_{i,j}$, initialize a list L of length n, where $L[i]$ gives the maximum value of the score function $2\sum_{X_{i,j}=1} C_{i,j} - \sum_{X_{i,j}=0} C_{i,j}$ while assuming $n = i+1$. At the lowest index $i = 0$, $L[0]$ stores the maximum value of the score function under the assumption $n = 1$ while at the largest index, $i = 0$, $L[n-1]$ stores the maximum value of the score function for the entire set of $X_{i,j}$.

Initialize an $n \times n$ matrix S to store the score function value of setting $C_{i,j}$ to 1 within a given right isosceles triangle $S_{i,j}$, where the right isosceles triangle $S_{i,j}$ is oriented such that its hypotenuse spans $M[i][i]$ and $M[j][j]$. Note in this case, $S_{i,j}$ is both the score function value of setting the $C_{i,j}$ to 1 within the right isosceles triangle as well as the notation that defines the specific right isosceles triangle whose values we are setting to 1.

Initialize a list A of length n, where we will store tuples of the right isosceles triangles $S_{i,j}$ whose values we wish set to 1 based on the maximum value of the score function for each n. For example, $A[i]$ stores the $S_{i,j}$ whose values we wish set to 1 based on the maximum value of the score function assuming $n = i + 1$.

Finally, initialize a matrix an $n \times n$ matrix C assigning values for each $C_{i,j}$ based on the $S_{i,j}$ found in A.

Recurrence:

To describe our recurrence constructing L, we start first with our base case. In the case of $L[0]$, we examine $X_{0,0}$ in M and determine its value. If $X_{0,0} = 1$, we set $C_{0,0} = 1$, and thus $L[0] = 2$ and $A[0] = (S_{0,0})$. Otherwise, if $X_{0,0} = 0$, we set $C_{0,0} = 0$, and thus $L[0] = 2$ and $A[0] = ()$ as we are not setting any values to 1.

Proceeding, our recurrence for constructing L is then the following:

$$L[i] = \max(L[i-1], L[i-1] + S_{i,i}, L[i-2] + S_{i-1,i} \ldots, L[0] + S_{1,i}, S_{0,i}).$$

Thus, the maximum value of $L[i]$ is either $L[i-1]$, $L[i-j] + S_{i-j+1,i}$, or $S_{0,i}$. Thus, for $A[i]$ we have:

$$A[i] = \begin{cases} A[i-1] & \text{if } L[i] = L[i-1] \\ A[i-j] + S_{i-j+1,i} & \text{if } L[i] = L[i-j] + S_{i-j+1,i} \\ S_{0,i} & \text{if } L[i] = S_{0,i}, \end{cases}$$

We continue recurring through all values of i till we have $L[n-1]$ and $A[n-1]$ and thus, have covered our entire set of $X_{i,j}$. Finally, we examine A, specifically $A[n-1]$, which holds the $S_{i,j}$ whose values should be set to 1 in order to maximize our score function. Thus, we iterate through C, assigning $C_{i,j}$ values to 1 based on $A[n-1]$ and assigning all other $C_{i,j}$ to 0. Finally we return C, which gives the assignments of $C_{i,j}$ for each pair (i, j) such that the score function is maximized and conditions (a) and (b) are satisfied.

Space:

The space complexity of the matrices and lists described above are as follows:

M is an $n \times n$ matrix requiring $O(n^2)$ space
L is a list of length n requiring $O(n^2)$ space.
A is a list of length n requiring $O(n)$ space.
S is an $n \times n$ matrix requiring $O(n^2)$ space.
C is an $n \times n$ matrix, requiring $O(n^2)$ space.

As such, the overall space complexity of our algorithm is $O(n^2)$.

Runtime:

To analyze the runtime complexity of our algorithm, we will analyze the runtime it takes to compute a unit of space for each data structure described above, then multiply by the corresponding space complexity to generate the overall runtime.

First, assigning each of the $O(n^2)$ entries in M takes $O(1)$ time, since we are simply assigning each entry the value 0 or 1 according to $X_{i,j}$. As such, the runtime of constructing M is $O(n^2)$.

Computing each entry of S takes $O(1)$ time using the following method: for all $i \in [0, n-1]$, calculate the value of all $S_{i,i}$, which is the score function value of setting $C_{i,i} = 1$. This gives us the score function value for all of the n triangles composed of a single entry along the diagonal. Then, calculate all $S_{i-1,i}$, by adding $S_{i-1,i-1}$ and $S_{i,i}$ to the score function value of setting $C_{i-1,i} = 1$. Continue to construct S in this manner by summing or subtracting smaller triangles using the score function value of setting a given $C_{i,j}$ to 1. Since we have shown this take $O(1)$ time, the runtime of constructing M is $O(n^2)$.

Computing each entry of the list L takes $O(n)$ time. For $L[n-1]$ (the value that requires the most comparisons to compute), we compare $n+1$ values of the form $L[i-j] + S_{i-j+1,i}$. Since we've already computed each preceding entry of L and all entries of S, accessing any of these $L[i-j] + S_{i-j+1,i}$ takes $O(1)$. Next, comparing n of these expressions to compute $L[n-1]$ takes $O(n)$. Thus, since there n values in L, constructing L takes $O(n^2)$ time.

Assigning each list value of A takes $O(1)$ time. For the $L[i-j] + S_{i-j+1,i}$ yielding the largest score function value for $L[i]$, we are simply storing the corresponding $A[i-j], S_{i-j+1,i}$ in A. Thus, constructing A takes overall $O(n)$ time.

Populating the matrix C takes $O(n^2)$ time since we are simply assigning each entry the value 0 or 1 according to $A[n-1]$. As such, the runtime of constructing C is $O(n^2)$.

Thus, the overall runtime of our algorithm is $O(n^2)$.

Proof of Correctness:

To prove our algorithm is correct, we must show that our recursion will give the assignment that maximizes the score function when assuming $n = i$. To demonstrate this, we can show that our algorithm considers the possible forms which our assignment can take and selects the optimal one for maximizing the score function value.

For the assignment which does not require any additional right isosceles triangles assigned in the ith column, our optimal assignment for maximizing the score function value has already been assigned in an earlier recursion, namely, when creating the term $L[i-1]$. As such, using the the term $L[i-1]$ gives the correct solution as we may assume that all preceding optimal assignments from $L[0]$ through $L[i-1]$ have been executed correctly when calculating the optimal assignment for $L[i]$. Note that this form includes the solution in which no triangles are assigned as the $L[i-1]$ assignment propagates down through smaller problems to create this solution.

For the assignment which requires a right isosceles triangle in the ith column, the assigned triangle could be of any size from from $S_{i,i}$ to $S_{0,i}$. In determining which of these triangles to include, we consider the score function value of assigning each triangle given by S as well as maximum score function value assignment for the corresponding part of the matrix left over, which has have solved by computing earlier entries of L. Comparing all such options, we find the combination of $L[i-j]+S_{i-j+1,i}$, or simply $S_{0,i}$ which maximizes the score function value and select it for $L[i]$.

These two forms cover all possible valid assignments of $C_{i,j}$ to 1 in accordance to (a) and (b). This is because our constraints dictate that $C_{i,k}=1 \iff C_{i,j}=1, C_{j,k}=1$ for $i \leq j \leq k$. As such, for any given $C_{i,k}=1$, all cells to the left and below must also be 1, creating a right-isosceles-triangle configuration where the hypotenuse is along the diagonal of the matrix. Thus, any valid assignment is of this form. Thus, we have proven the validity of $L[n-1]$ and $A[n-1]$ as the correct maximum score function value and its corresponding assignment. Using the triangles described by $A[n-1]$ and sub-indices of A encoded within $A[n-1]$ we may then assign $C_{i,j}$ values correctly to 1. That is, if $A[i]=A[i-j]+S_{i-j+1,i}$, we will set all $C_{i,j}$ within $S_{i-j+1,i}$ to 1, and examine $A[i-j]$ in the same way to determine the next values within C to set to 1, recursively doing this until the base case. way until we reach the base case. We assign all other $C_{i,j}$ within C to be 0. Thus, when we return C, we are returning the assignments of all $C_{i,j}$ such that the score function value is maximized and all constraints are met, so our algorithm is correct.