CS 124 Homework 4: Spring 2024

Anusha Murali

Collaborators: None

No. of late days used on previous psets: 2
No. of late days used after including this pset: 4

Homework is due Wednesday Mar 6 at 11:59pm ET. Please remember to select pages when you submit on Gradescope. Each problem with incorrectly selected pages will lose 5 points.

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 your papers to turn them in.

Collaboration Policy: You may collaborate on this (and all problem sets) only with other students currently enrolled in the class, and of course you may talk to the Teaching Staff or use Ed. You may also consult the recommended books for the course and course notes linked from the timetable. You may not use generative AI or large language models, or search the web for solutions, or post the questions on chat forums. Furthermore, you must follow the "one-hour rule" on collaboration. You may not write anything that you will submit within one hour of collaborating with other students or using notes from such sources. That is, whatever you submit must first have been in your head alone, or notes produced by you alone, for an hour. Subject to that, you can collaborate with other students, e.g. in brainstorming and thinking through approaches to problem-solving.

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 no credit. Again, try to make your answers clear and concise.

Problems

1. Let $n \in \mathbb{N}$. An *n-subset-dict* $\mathbf{x} = \{x_S\}_{S \subseteq \{1,\ldots,n\}}$ is any collection of 2^n bits indexed by all possible subsets of $\{1,\ldots,n\}$. The *polar transform* P_n is a function that maps n-subset-dicts to n-subset-dicts as follows: if $\mathbf{w} = P_n(\mathbf{x})$, then for every $S \subseteq \{1,\ldots,n\}$,

$$w_S = \sum_{T : T \subseteq S} x_T \pmod 2. \tag{1}$$

   (Note that "sum (mod 2)" is the same as XOR. So $a+b+c+d \pmod 2$ is the same as $a \oplus b \oplus c \oplus d$. So if you are more familiar with the XOR operation, you can think of the rest of the question in terms of that.)

Example. Suppose $n = 2$. Then $\mathbf{x} = \{x_\emptyset, x_{\{1\}}, x_{\{2\}}, x_{\{1,2\}}\} = \{1,1,0,1\}$ is a 2-subset-dict. The polar transform of this is given by $\mathbf{w} = \{w_\emptyset, w_{\{1\}}, w_{\{2\}}, w_{\{1,2\}}\} = P_2(\mathbf{x})$, where

$$w_\emptyset = x_\emptyset = 1 \tag{2}$$
$$w_{\{1\}} = x_\emptyset + x_{\{1\}} \pmod 2 = 0 \tag{3}$$
$$w_{\{2\}} = x_\emptyset + x_{\{2\}} \pmod 2 = 1 \tag{4}$$
$$w_{\{1,2\}} = x_\emptyset + x_{\{1\}} + x_{\{2\}} + x_{\{1,2\}} \pmod 2 = 1 \tag{5}$$

(a) **(15 points)** Design an $O(n \log n)$ time algorithm to compute the polar transform, i.e., to compute $P_n(\mathbf{x})$ given n-subset-dict $\mathbf{x} = \{x_S\}_{S \subseteq [n]}$.

Solution: The polar transform $P_n(x)$ maps n-subset-dicts to n-subset-dicts such that if $w = P_n(x)$, then for every $S \subseteq \{1, \ldots, n\}$,

$$w_S = \sum_{T : T \subseteq S} x_T \pmod 2 .$$

We note that the above operation is equivalent to finding the bitwise subset of $\mathbf{x}$, where $\mathbf{x} = \{x_S\}_{S \subseteq \{1,\ldots,n\}}$. We first explain the idea of finding the subsets of $\mathbf{x}$ using bitwise operation using the example given in the problem before providing an $O(n \log n)$ algorithm.

Let $n = 2$, so $\{x_\emptyset, x_{\{1\}}, x_{\{2\}}, x_{\{1,2\}}\} = \{1,1,0,1\}$. Let the keys of these values in the dictionary be $x = 0,1,2,3$. We can find the corresponding $P_2(x)$ with the help of bit-masks as follows. For all possible values of $x = 0, \ldots 3$, we will check all the bitwise subsets for $i = 0 \ldots 3$. Specifically we check whether $(x \,\&\, i)$ is equal to i. If $(x \,\&\, i)$ is equal to i, then we will include that element in the set. We omit $\pmod 2$ in the following operations (and apply it at the end) for the sake of simplicity. We assume x as the mask in the discussion below.

$\underline{x = 0}$: For $i = 0$, $(x \,\&\, i) = i$. For $i = 1,2,3$, $(x \,\&\, i) \neq i$. So $P_2(0) = x_\emptyset = 1$.

$\underline{x = 1}$: For $i = 0$, $(x \,\&\, i) = i$, and for $i = 1$, $(x \,\&\, i) = i$. For $i = 2,3$, $(x \,\&\, i) \neq i$. So $P_2(1) = x_\emptyset + x_{\{1\}} = 1 + 1 = 2$.

$\underline{x = 2}$: For $i = 0$, $(x \,\&\, i) = i$, and for $i = 2$, $(x \,\&\, i) = i$. For $i = 1,3$, $(x \,\&\, i) \neq i$. So $P_2(2) = x_\emptyset + x_{\{2\}} = 1 + 0 = 1$.

$\underline{x = 3}$: For all $i = 0,1,2,3$, $(x \,\&\, i) = i$. So $P_2(3) = x_\emptyset + x_{\{1\}} + x_{\{2\}} + x_{\{1,2\}} = 1 + 1 + 0 + 1 = 3$.

Taking $\pmod 2$ of $P_2(0), P_2(1), P_2(2), P_2(3)$, we find $P_2(\mathbf{x}) = \{1,0,1,1\}$ as given in the problem.

The above approach has a runtime of $O(2^n \cdot 2^n) = O(2^{2n})$, as we are iterating through each of the 2^n values of i for each of the 2^n values of x. This can be significantly optimized by making use of an observation to reduce the number of comparisons, as well as saving the bitwise subsets as we compute them and then reusing them whenever we need them. Specifically, we make the following observations:

i. Suppose the i-th bit of the mask x is 0. In this case, the elements in this set can only differ in the first $i-1$ bits. This is because, when a number has a 1 at the i-th bit, while the mask x has a 0, the number cannot be part of the subset. Therefore, it must be that $S(x,i) = S(x,i-1)$.

ii. Suppose the i-th bit of the mask x is 1. In this case, the elements in this set can be divided into two distinct sets: (a) numbers whose i-th bit is 1, which differ from mask x in the

subsequent $i-1$ bits, and (b) numbers whose i-th bit is 0, which differ from $(x \oplus 2^i)$ in the subsequent $i-1$ bits. Therefore, the i-th bit of the mask x is 1, then it must be that $S(x,i) = S(x,i-1) \cup S(x \oplus 2^i, i-1)$.

Hence, we can compute $S(x,i)$ using the following recursive equation:

$$S(x,i) = \begin{cases} S(x,i-1) & \text{if } i\text{-th bit is 0} \\ S(x,i-1) \cup S(x \oplus 2^i, i-1) & \text{if } i\text{-th bit is 1.} \end{cases} \tag{6}$$

Our algorithm is therefore as follows. It takes an n-subset-dict A and n as inputs and returns $P_n(A)$.

Algorithm 1 POLAR-TRANSFORM(A, n)

```
1: if n == 0 then
2:     return A
3: P = [0, ..., 0]                                   {Initialize all 2^n elements of P to 0}
4: table = [[0 for i in range(n)] for j in range(2^n)]   {2^n × n table to hold intermediate results}
5: for x = 0 to 2^n − 1 do
6:     for i = 0 to n − 1 do
7:         if (x & (1 << i)) then
8:             if (i == 0) then
9:                 table[x][i] = A[x] + A[x ⊕ 1]                {When i = 0, this is a base case / leaf}
10:            else
11:                table[x][i] =table[x][i − 1] + table[x ⊕ (1 << i)][i − 1]      {From Equation 6}
12:        else
13:            if (i == 0) then
14:                table[x][i] = A[x] .                         {When i = 0, this is a base case / leaf}
15:            else
16:                table[x][i] =table[x][i − 1]                                  {From Equation 6}
17:    P[x] =table[x][n − 1] (mod 2)                        {Return the results in (modulo 2)}
18: return P
```

Correctness: We show the correctness using induction. Our algorithm uses bit-masks to select the subsets. Specifically, when there are n elements, we represent all the possible subsets of the elements using $0, \ldots, 2^n - 1$ in binary form. The elements whose positions have a 1 are included in the subset, and whose positions have 0 are excluded from the subset.

Basis Step: For an empty set, $w_\emptyset = x_\emptyset$. The algorithm correctly returns $w_0 = P[0] = A[0] = x_\emptyset$ on line number 2.

Inductive Step: Suppose that the algorithm outputs the correct subsets for $P_n(x)$, implying that the following recursion holds for all the values of x from $0, \ldots, 2^n$ and all the values of i from $0, \ldots, n$:

$$S(x,i) = \begin{cases} S(x,i-1) & \text{if } i\text{-th bit is 0} \\ S(x,i-1) \cup S(x \oplus 2^i, i-1) & \text{if } i\text{-th bit is 1.} \end{cases}$$

For $n+1$, we consider the following two cases: (a) The i-th bit of $x = 0$. In this case, regardless of whether the number has 0 or 1 at its i-th bit, $x \,\&\, y = 0$ and consequently the number

cannot be part of the subset. So, it must be that $S(x,i) = S(x,i-1)$. (b) The i-th bit of $x = 1$. In this case, we can divide the elements into two distinct sets, where the first set contains numbers whose i-th bit is 1, which differ from mask x in the subsequent $i-1$ bits, and the second set contains numbers whose i-th bit is 0, which differ from $(x \oplus 2^i)$ in the subsequent $i-1$ bits. The XOR ($\oplus$) operation is used to identify the positions where the mask and number differ. Therefore, the i-th bit of the mask x is 1, then it must be that $S(x,i) = S(x,i-1) \cup S(x \oplus 2^i, i-1)$. Therefore, assuming the inductive hypothesis that the recursion is true for $P_n(x)$, the recursion holds for $P_{n+1}(x)$.

Runtime: The outer for loop on line number 5 iterates 2^n times over the possible masks. The inner for loop on line number 6 iterates n times. Assuming the table lookups and the XOR operations take $O(1)$ time, the overall runtime of the above dynamic programming algorithm is $O(n2^n)$. Letting N be the number of elements, $N = 2^n$. Therefore the runtime, in terms of N is $O(N \log N)$ as desired. □

(b) **(10 points)** Design an $O(n \log n)$ time algorithm to *invert* the polar transform. That is, given n-subset-dict $\mathbf{w} = \{w_S\}_{S \subseteq [n]}$, the goal is to compute $\mathbf{x} = \{w_S\}_{S \subseteq [n]}$ for which $\mathbf{w} = P_n(\mathbf{x})$.

Solution: We claim that the algorithm used in Part (a) to compute the polar transform of an n-subset-dict can be used to invert the polar transform. We first provide a more simplified version of the POLAR-TRANSFORM() algorithm from Part (a). This is achieved by initializing $P_n(x)$ to $A(x)$ and reversing the order in which we check the bit-masks to select the subsets. The algorithm otherwise remains the same.

Algorithm 2 POLAR-TRANSFORM-SIMPLIFIED(A, n)

```
1: for i = 0 to 2^n − 1 do
2:     P[i] = A[i]                                   {Initialize P to A}
3: for i = 0 to n − 1 do
4:     for x = 0 to 2^n − 1 do
5:         if (x & (1 << i)) then
6:             P[x] = P[x] + P[x ⊕ (1 << i)]         {From Equation 6}
7: for x = 0 to 2^n − 1 do
8:     P[x] = P[x] (mod 2)                           {Return the results in (modulo 2)}
9: return P
```

Correctness: We claim that the algorithm used in Part (a) (which is essentially the above POLAR-TRANSFORM-SIMPLIFIED() algorithm above) to compute the polar transform of an n-subset-dict can be used to invert the polar transform.

In other words, we want to show that $w \circ w(x) = x$ in modulo 2. We inductively (on the size of the indexing subsets) prove this as follows.

Basis Step: For indexing-subset size $k = 0$, $w_\emptyset = x_\emptyset$, which means $w(w_\emptyset) = w(x_\emptyset) = x_\emptyset$, and therefore $w \circ w(x) = x$.

Inductive Step:

Assume that $w \circ w(x) = x$ in modulo 2 is true for all subset sizes $1, \ldots, k-1$ (where $k \geq 1$). Now, consider subset size of k. So (WLOG), we have $w_{\{1,2,\ldots,k\}} = x_\emptyset + x_{\{1\}} + x_{\{2\}} + \cdots + x_{\{1,2,\ldots,k\}}$. Since

the number of subsets involving k distinct elements is 2^k, the number of expressions on the RHS of the above expression is even. Therefore, we find,

$$\begin{aligned} w\left(w_{\{1,2,\ldots,k\}}\right) &= w\left(x_\emptyset + x_{\{1\}} + x_{\{2\}} + \cdots + x_{\{1,2,\ldots,k\}}\right) \\ &= w(x_\emptyset) + w(x_{\{1\}}) + w(x_{\{2\}}) + \cdots + w(x_{\{1,2,\ldots,k\}}) \\ &= x_\emptyset + (x_\emptyset + x_{\{1\}}) + (x_\emptyset + x_{\{2\}}) + \cdots + (x_\emptyset + (x_\emptyset + x_{\{1\}}) + (x_\emptyset + x_{\{2\}}) + \\ &\quad \cdots + (x_\emptyset + x_{\{1\}} + x_{\{2\}} + \cdots + x_{\{k\}}) \\ &= 2^k x_\emptyset + 2^{k-1} x_{\{1\}} + 2^{k-1} x_{\{2\}} + \cdots + 2^{k-2} x_{\{1,2\}} + \cdots + 2^{k-3} x_{\{1,2,3\}} + \cdots + 2^0 x_{\{1,2,\ldots,k\}} \end{aligned}$$

Since $P_n(x)$ returns the answer in modulo 2, taking modulo 2 of the RHS, we note that all the terms on the RHS, except the last one, disappear. Therefore, we obtain,

$$\begin{aligned} w\left(w_{\{1,2,\ldots,k\}}\right) &= 0 + 0 + \cdots + x_{\{1,2,\ldots,k\}} \\ &= x_{\{1,2,\ldots,k\}}, \end{aligned}$$

which implies that $w \circ w(x) = x$ in modulo 2 is true for subset size k. Therefore, we have shown that our original algorithm derived in Part (a), when $\mathbf{w} = P_n(\mathbf{x})$ is given as its input, will invert the polar transform and output the original $\mathbf{x} = \{w_S\}_{S \subseteq [n]}$. □

Note: Suppose we are not returning $P_n(\mathbf{x})$ in modulo 2 and are interested in finding out the original n-subset-dict, $\mathbf{x}$. We note that the algorithm POLAR-TRANSFORM-SIMPLIFIED() is analogous to the algorithm for generating prefix-sums. In order to obtain the initial values resulting in the prefix sum, all what we need to do is to change line number 6 in POLAR-TRANSFORM-SIMPLIFIED() to the following, where we have replaced the '+' with a '−':

$$P[x] = P[x] - P[x \oplus (1 << i)].$$

This modification to POLAR-TRANSFORM-SIMPLIFIED() will return the initial n-subset-dict, $\mathbf{x}$, even if we are not working in modulo 2. □

Runtime: Since the original algorithm POLAR-TRANSFORM() can be used to find the initial n-subset-dict, as shown in Part (a), the runtime is $O(\mathrm{N}\log\mathrm{N})$, where $N = 2^n$. □

2. Let $L = \{\mathbf{z}_1, \ldots, \mathbf{z}_n\}$ be a list of n points in d dimensions. We say that a point $\mathbf{z}_i$ *covers* a point $\mathbf{z}_j$ if for every $1 \leq \ell \leq d$, the ℓ-th coordinate of $\mathbf{z}_i$ is at least as large as the ℓ-th coordinate of $\mathbf{z}_j$. Further suppose for simplicity that given any two real numbers, we can compare whether one is larger than the other in time $O(1)$. Additionally, you may assume that all of the entries across all of the vectors $\mathbf{z}_1, \ldots, \mathbf{z}_n$ are distinct.

   (a) **(10 points)** Suppose $d = 2$. Design an $O(n \log n)$-time algorithm which, given this list L, outputs a list of numbers $a_1, \ldots, a_n$, such that for every $1 \leq i \leq n$, a_i is the number of points that $\mathbf{z}_i$ covers. (Hint: you may find it helpful to consult the Lecture 9 notes on closest pair)

   Solution: The algorithm is a modification of MERGE-SORT, along with one application of MERGE-SORT itself. We first run MERGE-SORT on $L = \{\mathbf{z}_1, \ldots, \mathbf{z}_n\}$ according to the first coordinate to get L^*. Then we approach L^* with a divide-and-combine paradigm. The idea is that for two equal-sized parts of L^*, we assume that both parts have also been (1) sorted in increasing order according to the 2^{nd} coordinates, and (2) all the respective a_i's have been correctly computed thus far (without taking into account the members from the other parts). To explain (2), we mean that if we have parts A_1 and A_2, then the a_i values of the members of A_2 have been correctly determined only with respect to the other members in A_2, and vice versa for A_1. We will show that at the end of the merging step, all the members of the result of merging A_1 and A_2 will satisfy both (1) and (2) again. Note that the push(L, x) function pushes x at the beginning of the array L. Note that in the following, we change the order of the elements returned, but they can be easily maintained by also additionally storing the original indices, along with the a_i values, and then returning everything in the original order using those indices using an additional $O(n)$ step. We omit that for clarity. Also, note that our indexing starts at 1, instead of 0.

   The following is the main function.

Algorithm 3 MODIFIED-MERGE-SORT

Run MERGE-SORT(L) based on the first coordinate to get L^*
Set $\hat{L}, A \leftarrow$ CREATE-A(L^*)
Return $\hat{L}, A$

The CREATE-A(L^*) function is shown on the next page.

Algorithm 4 CREATE-A(L^*)

if $|L| = 1$ **then**
  return $L, [1]$
Let $L_1 \leftarrow L[1, \ldots, n/2]$ and $L_2 \leftarrow L[n/2+1, \ldots, n]$
Set $\hat{L}_1, A_1 \leftarrow$ CREATE-A(L_1) and $\hat{L}_2, A_2 \leftarrow$ CREATE-A(L_2)
Let $\hat{L} \leftarrow [\,], A \leftarrow [\,]$
Let $i_1 \leftarrow n/2, i_2 \leftarrow n$
while $i_1 \geq 1$ and $i_2 \geq n/2+1$ **do**
  if $\hat{L}_2[i_2](2) \geq \hat{L}_1[I_1](2)$ **then**
    $A_2[i_2] \leftarrow A_2[i_2] + i_1$
    push($\hat{L}, \hat{L}[i_2]$) and push($A, A_2[I_2]$)
    Set $i_2 \leftarrow i_2 - 1$
  else
    push($\hat{L}, \hat{L}_1[i_1]$) and push($A, A_1[I_1]$)
    Set $i_1 \leftarrow i_1 - 1$
if $i_1 \geq 1$ or $i_2 \geq n/2+1$ **then**
  Let $i \leftarrow i_1$ if first condition holds, otherwise $i \leftarrow i_2$
  Let $A' \leftarrow A_1$ if first condition holds, otherwise $A' \leftarrow A_2$
  Let $L' \leftarrow \hat{L}_1$ if first condition holds, otherwise $L' \leftarrow \hat{L}_2$
  Let $\tau \leftarrow 1$ if first condition holds, otherwise $\tau \leftarrow n/2+1$
  while $i \geq \tau$ **do**
    push($\hat{L}, L'[i]$) and push($A, A'[I]$)
    Set $i \leftarrow i - 1$
Return $\hat{L}, A$

Correctness: We prove the correctness of the algorithm via strong induction. We can think of this recursive algorithm as a binary tree, where each node is representing computations on disjoint and independent sections of the original array L. Here, the induction will be on the level of the tree k, where the level of the leaves is $k = 1$, and the level of the root is $k = \log(n)$. We claim that before starting the computations in any node on level k, the following two conditions for the sub-tree of that node must hold.

i. L_1, L_2 must be sorted in increasing order of the second coordinates of their members.
ii. A_1, A_2 must have "correct" a_i values only with respect to their respective members. In other words, for any $x \in L_1$, it's corresponding a in A_1 must show the exact number of elements in L_1 that x covers, and vice versa for any member in L_2.

We will then show this for level $k+1$, as well, by showing that $\hat{L}$ and A satisfy the correct properties by the end of the computations in level k.

Basis Step. At the end of level 1, which only has singleton arrays, the elements are only getting compared with themselves, so they trivially cover themselves. Since the algorithm is just returning those singleton arrays with 1 as their a_i values by construction, the algorithm works correctly at this level, and we satisfy the claim at the end of level 1, and hence, at the beginning of level 2.

Inductive Step. The proof is very similar to that for MERGE-SORT's correctness. Therefore, will omit the proof of correctness for the sorting part, and will focus on the correctness of A.

Note that at the beginning, when L gets divided into L_1 and L_2, the first coordinates of all the members of L_2 are larger than those of all the members of L_1 because we had started off the main algorithm by sorting L.

Now, we know (by our inductive hypothesis) $\hat{L}_1$ and $\hat{L}_2$ are sorted within in increasing order of the second coordinates. In the first WHILE-loop, (we can also argue this via induction on the iterations of the loop) if the second coordinate of a point in $\hat{L}_2$ is larger than the second coordinate of a point in $\hat{L}_1$, then the former covers the latter, and because $\hat{L}_1$ and $\hat{L}_2$ are sorted in increasing order, the first time when that happens, it must be the case that the point in $\hat{L}_2$ must cover the said point in $\hat{L}_1$ and all the points before it. This is because the first coordinate of every point in $\hat{L}_2$ is also larger than the first coordinate of every point in $\hat{L}_1$.

On the other hand, if the second coordinate of a point in $\hat{L}_2$ is smaller than that of a point in $\hat{L}_1$, it means that the former does not cover the latter, and it must be the case that no point in $\hat{L}_1$ that comes after the said point in $\hat{L}_1$ is covered by that point in $\hat{L}_1$. Therefore, we will have to look at a point in $\hat{L}_1$ that has a smaller second coordinate, which is what the algorithm does by decreasing i_1 by 1.

This means that at the end of the first WHILE-loop, we have updated the a_i values of all the points in L in that node only with respect to L. Due to the guarantees of MERGE-SORT, $\hat{L}$ is still sorted in increasing order of the second coordinates. Note that the above argument within this inductive step could also be formalised via induction, but it would be very similar to that for the correctness of MERGE-SORT.

Runtime: The running time is $O(n\log(n))$ by the exact same analysis of MERGE-SORT. This is because we just introduce a few extra $O(1)$ steps within the WHILE-loops in the recursion.

(b) **(0 points, optional)** Generalize your algorithm and analysis in the previous part to give an $O(n\log^{d-1} n)$-time algorithm for any constant d.

3. (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 recommended line length is M. We define a measure of neatness as follows. The extra space on a line (using one space between words) containing words i through j is $A = M - j + i - \sum_{k=i}^{j} \ell_k$. (Note that the extra space may be negative, if the length of the line exceeds the recommended length.) The penalty associated with this line is A^3 if $A \geq 0$ and $2^{-A} - A^3 - 1$ if $A \leq 0$. The penalty for the entire paragraph is the sum of the penalty over all the lines, except that the last line has no penalty if $A \geq 0$. Variants of such penalty function have been proven to be effective heuristics 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. We first formulate the following definitions that will be used in our algorithm:
(1) The number of extra spaces on a line containing words i through j:

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

(2) The penalty associated with a line, which has words i through j, due to extra spaces:

$$\text{linePenalty}[i, j] = \begin{cases} A^3 & \text{if } A \geq 0 \\ 2^{-A} - A^3 - 1 & \text{if } A < 0 \quad \text{(exceeded the margin)} \\ 0 & \text{if } j = n \text{ and } A \geq 0 \quad \text{(this is the last line)} \end{cases}$$

We note that the linePenalty is always non-negative (i.e: linePenalty ≥ 0). Our goal is to minimize the total linePenalty (summing over all the lines) of the entire paragraph. We can achieve this using dynamic programming by making the following observation:
Assume that the words $1, 2, \ldots, j$ have been optimally arranged. Let us say that the last line of this optimal arrangement has words i through j. Therefore, the preceding lines contain words $1, 2, \ldots, i-1$, which by our assumption, are optimally arranged.
Let us define cost$[j]$ to be the cost of such an optimal arrangements of the words $1, 2, \ldots, j$. Suppose that the last line has words i through j, then

$$\text{cost}[j] = \text{cost}[i-1] + \text{linePenalty}[i, j].$$

As the basis case, let cost[0] = 0. Therefore, we have the following recursive definition.

(3) The cost of optimally arranging words $1, 2, \ldots, j$

$$\text{cost}[j] = \begin{cases} 0 & \text{if } j = 0, \\ \min\limits_{1 \leq i \leq j} \left(\text{cost}[i-1] + \text{linePenalty}[i, j] \right) & \text{if } j > 0. \end{cases} \tag{7}$$

The above recursive definition of the cost of the optimal solution has the overlapping sub-problem property and is therefore the motivation for our dynamic programming algorithm since it makes use of the optimal solutions found for the sub-problems. This allows us to minimize the sum of linePenalty over all lines in the entire paragraph.

Correctness: We can inductively show the correctness as follows. The base case for the cost is cost[0] = 0. Suppose we have an optimal arrangements for the words $1, 2, \ldots, j$. Let us say that the last line has words i through j. If the last line has an optimal arrangement, then from our recursion formula given in Equation 7 and a straightforward cut-and-paste argument, all the previous lines containing the words $1, 2, \ldots, i-1$ must have optimal arrangements. This argument repeats until we reach $c[0]$, for which the cost is 0.

The algorithm is defined below.

Algorithm 5 PRINT-PARAGRAPHS(words, M)

```
1:  n = length(words)
2:  if n = 0 then
3:      return                                   {Number of words = 0, so just return}
4:  Initialize all entries of A[i, j] = 0, linePenalty[i, j] = 0, cost[j] = ∞   (1 ≤ i, j ≤ n)
5:  Initialize all entries of pList[j] = 0       {pList keeps track of which line each cost value came from}
6:  Initialize cost[j] = 0       {The cost array has n + 1 entries, with the 0-th entry initialized to 0}
7:  {The following nested for loop computes the extra spaces for all words from 0 to n-1}
8:  for i = 1 to n do
9:      A[i][i] = M−len(words[i])
10:     for j = i + 1 to n do
11:         A[i][j] = A[i][j − 1]−len(words[j]) −1
12: {The following nested for loop computes the line costs}
13: for i = 1 to n do
14:     for j = i to n do
15:         if A[i][j] < 0 then
16:             linePenalty[i][j] = 2^{−A[i][j]} − A[i][j]^3 − 1
17:         else if j = n and A[i][j] ≥ 0 then
18:             linePenalty[i][j] = 0                          {This is the last line}
19:         else
20:             linePenalty[i][j] = (A[i][j])^3
21: cost[0] = 0                                    {Cost for the base case}
22: {The following nested for loop computes the optimal costs (values) for each line}
23: for j = 1 to n) do
24:     for i = 1 to j) do
25:         thisCost = cost[i − 1] + linePenalty[i][j]
26:         if cost[j] > thisCost then
27:             cost[j] = thisCost
28:             pList[j] = i                   {pList keeps track of which line each cost value came from}
```

Improvement of PRINT-PARAGRAPHS(): The above algorithm can be further improved by observing that at most $\lceil M \rceil$ words can be accommodated in a single line, as there has to be a space between two words, even if the words are only one character long. This implies, we only need to compute $A[i, j]$ and linePenalty$[i, j]$ only when the number of words in a line $\leq \lceil M \rceil$. In addition, the observation also allows us to reduce the number of execution of the for loop on line number 24 from $\max(1, j - \lceil M \rceil + 1)$ to j.

Runtime: The three doubly-nested for loops, the first one for computing the extra spaces,

the second one for computing the line penalties, and the last one for computing the optimal costs of the lines, in the above algorithm have a runtime of $O(n^2)$. Therefore, the overall runtime is $O(n^2)$. We can improve this runtime by making changes to the for loops computing $A[i, j]$ and linePenalty$[i, j]$ such that their body is executed only when $j - i + 1 \leq \lceil M \rceil$, as well as changing the for loop on line number 24 to run from $\max(1, j - \lceil M \rceil + 1)$ to j. This reduces the runtime from$O(n^2)$ to $\Theta(nM)$.

Note: Instead of storing $A[i, j]$ and linePenalty$[i, j]$, by computing them on the fly as they are needed, the space can be reduced to $\Theta(n)$.

(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 as part of the length of a word, so the last word of the question has length eleven, which counts 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 problem 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:

i. $\underline{M = 40}$: The minimal penalty is 396. The optimal division words into lines is shown below:

```
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 as part
of the length of a word, so the last
word of the question has length eleven,
which counts 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 problem 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.)
```

ii. $\underline{M = 72}$: The minimal penalty is 99. The optimal division words into lines is shown below:

```
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 as part of the
length of a word, so the last word of the question has length eleven, which
counts 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 problem
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.)
```

4. **(25 points)** At the local library, every one of the n books in the inventory is labeled with a real number; denote these numbers by $x_1 \leq \cdots \leq x_n$. The complement $\mathbb{R}\backslash\{x_1,\ldots,x_n\}$ decomposes into $n+1$ disjoint open intervals $I_0,\ldots,I_n$. We are given numbers $p_1,\ldots,p_n$ such that p_i is the probability that the book labeled with x_i is requested, as well as numbers $q_0,\ldots,q_n$ such that q_i is the probability that some number in the interval I_i is requested (corresponding to a book which is not available at the library). We assume that $p_1+\cdots+p_n+q_0+\cdots+q_n=1$.

   Our goal is to construct a binary search tree for determining whether a requested number z is among the n books in the inventory. The tree must have n internal nodes and $n+1$ leaves with the following properties:

   - Every internal node corresponds to a unique choice of x_i. At this node, one checks whether the requested number z is less than, greater than, or equal to x_i. If it is less (resp. greater) than x_i, one proceeds to the left (resp. right) child node. If it is equal to x_i, the search terminates and we conclude that book is available.
   - Every leaf node corresponds to a unique choice of I_j. If one arrives at a leaf node, the search terminates and we conclude that the book is unavailable.

   The goal is to construct a binary search tree that minimizes the *expected number of internal nodes queried*. Give an $O(n^3)$ algorithm for finding an optimal such binary search tree, given the numbers $p_1,\ldots,p_n,q_0,\ldots,q_n$.

   Solution: Our goal is to construct an optimal binary search tree (BST) that minimizes the expected number of internal nodes queried. Since there are $\binom{2n}{n}/(n+1)$ binary search trees with n nodes, we will use dynamic programming to find the optimal BST, which will minimize the overall search cost.

   We formulate the problem as follows. Let T be a BST, which has the value x_r at its root. The left subtree, T_L, of T, consists of values less than x_r and the right subtree, T_R, of T consists of values greater than x_r. Any subtree of our optimal BST will contain values in sorted order, $x_i,\ldots,x_j$, where $1 \leq i \leq j \leq n$. The subtree containing values $x_i,\ldots,x_j$ has leaves that represent the ranges $I_{i-1},\ldots,I_j$.

   The left subtree, T_L, of root x_r contains values $x_i,\ldots,x_{r-1}$ and the right subtree, T_R, contains values $x_{r+1},\ldots,x_j$. In addition, the leaves of the left subtree contains the ranges, $I_{i-1},\ldots,I_{r-1}$, and the leaves of the right subtree contains the ranges $I_r,\ldots I_j$. In this manner, we can recursively construct optimal subtrees from T_L and T_R.

   Let $C(i,j)$ represents the expected cost of searching the BST containing the set of values $\{x_k : i \leq k \leq j\}$. Our goal is to find the optimal BST that minimizes $C(1,n)$. When $j=i-1$, we obtain the base case with the range node I_{i-1}. The expected cost of searching in this case is $C(i,j)=C(i,i-1)=q_{i-1}$, which is the probability of the unsuccessful search ending up in this leave node. When $j \geq i$, the sum of probabilities of the subtrees rooted at x_r is,

$$w(i,j)=\sum_{k=i}^{j} p_k+\sum_{k=i-1}^{j} q_k.$$

Therefore, we can write the recursive equation to compute the subproblems as follows:

$$C(i,j) = \begin{cases} 0 & \text{if } i = j+1 \\ \min\limits_{i \le k \le j} \{C(i,k-1) + C(k+1,j) + w(i,j)\} & \text{if } i \le j \end{cases} \tag{8}$$

Our algorithm, as the base case, first computes the cost of the empty subtrees with leaves, $C(i,i-1)$. Then it computes the cost of the subtrees that contain one value, and then computes the cost of the subtrees that contain two adjacent values, each time computing the cost of the next larger tree with one more value of x_i. This recursion is based on the Equation 8 above.

Algorithm 6 OPTIMAL-BST

```
1:  for i = 1 to n+1 do
2:      C(i, i-1) ← q_i - 1                          {Initialize the base case cost}
3:      w(i, i-1) ← q_i - 1                          {Initialize the weights using the leaf probabilities}
4:  for k = 1 to n do
5:      for i = 1 to n - k + 1 do
6:          j ← i + k - 1
7:          C(i, j) ← ∞
8:          w(i, j) ← w(i, j-1) + p_j + q_j          {Update the probabilities}
9:          for m ← i to j do
10:             if C(i, m-1) + C(m+1, j) + w(i, j) < C(i, j) then
11:                 C(i, j) ← C(i, m-1) + C(m+1, j) + w(i, j)
12:                 root(i, j) ← m                   {Remember the index of the optimal subtree}
13: return root
```

Correctness: The above algorithm, OPTIMAL-BST is based on the recursive formula given by Equation 8. Therefore, we want to show that the recursive formula given by Equation 8 correctly computes, $C(i,j)$, which is the cost of an optimal BST containing the search values (and the ranges of missing values).

Consider a binary search tree, T, whose root is x_r. By the construction of T, the left subtree, T_L, contains values less than x_r and the right subtree, T_R, contains values greater than x_r. Let us say that we are looking for the value x_i, which can be found in one of the internal nodes of the BST with probability p_i or in the leaves with the probability q_{i-1}. Therefore, in either case, the probability of finding or not finding x_i is $w_i = p_i + q_{i-1}$. The cost of this operation for x_i in tree T is,

$$\sum_{i=1}^{n} w_i \cdot (d(i,T) + 1),$$

where $d(i,T)$ is the depth of the node corresponding to x_i in tree T. The number of comparisons is 1 more than the depth as the number of nodes is one more than the node's depth. Let $\mathscr{C}(T)$ be the cost of the tree rooted at x_i. Therefore, considering the costs of T_L, T_R and the root node, we can write $\mathscr{C}(T)$ as,

$$\mathscr{C}(T) = \sum_{k \in T_L} w_k \cdot (d(k,T) + 1) + \sum_{k \in T_R} w_k \cdot (d(k,T) + 1) + w_r,$$

where w_r is the probability of successfully finding or not finding the value x_r. Simplifying, we find,

$$\begin{aligned}
\mathscr{C}(T) &= \sum_{k \in T_L} w_k \cdot (d(k,T)+1) + \sum_{k \in T_R} w_k \cdot (d(k,T)+1) + w_r \\
&= \sum_{k \in T_L} w_k \cdot d(k,T) + \sum_{k \in T_R} w_k \cdot (d(k,T) + \sum_{k \in T} w_k \\
&= \sum_{k \in T_L} w_k \cdot (d(k,T_L)+1) + \sum_{k \in T_R} w_k \cdot (d(k,T_R)+1) + \sum_{k \in T} w_k \\
&= \mathscr{C}(T_L) + \mathscr{C}(T_R) + \sum_{k \in T} w_k.
\end{aligned}$$

This shows that the cost of our binary search tree T can be given in terms of the cost of its left and right subtrees as follows:

$$\mathscr{C}(T) = \mathscr{C}(T_L) + \mathscr{C}(T_R) + \sum_{k \in T} w_k.$$

Now, suppose T_{ij} the minimum cost BST that contains the values in the range $x_i, \ldots, x_j$, $1 \le i \le k \le j$, and has the value x_k at its root. Therefore, by the construction, the left subtree of T_{ij} is a BST with values from $x_i, \ldots, x_{k-1}$, and the right subtree of T_{ij} is a BST with values from $x_{k+1}, \ldots, x_j$. Given that T_{ij} is the minimum cost BST among all the possible BSTs that contain the values in the range $x_i, \ldots, x_j$ and the root x_k, from our derivation above ($\mathscr{C}(T) = \mathscr{C}(T_L) + \mathscr{C}(T_R) + \sum_{k \in T} w_k$), it must be that the left and right subtrees of T_{ij} must be optimal subtrees containing the values $x_i, \ldots, x_{k-1}$ and $x_{k+1}, \ldots, x_j$ respectively.

The cost of the above subtrees are $C(i, k-1)$ and $C(k+1, j)$ respectively. Hence, we find,

$$\begin{aligned}
\mathscr{C}(T) &= \mathscr{C}(T_L) + \mathscr{C}(T_R) + \sum_{k \in T} w_k \\
&= C(i, k-1) + C(k+1, j) + \sum_{k \in T} w_k.
\end{aligned}$$

Therefore, the cost of the BST that contains the values $x_i, \ldots, x_j$, which has the minimum cost is,

$$\begin{aligned}
C(i,j) &= \min_{i \le k \le j} \{\mathscr{C}(T_{ij})\} \\
&= \min_{i \le k \le j} \left\{C(i, k-1) + C(k+1, j) + w(i,j)\right\},
\end{aligned}$$

which proves that the recursive formula given by Equation 8 correctly computes, $C(i, j)$, which is the cost of an optimal BST containing the search values (and the ranges of missing values). □

Runtime: Our algorithm uses three nested for loops (on line numbers 4, 5, and 9), where each loop runs at most n times yielding a runtime of $O(n^3)$. □