CS 124 Homework 4: Spring 2024

Collaborators: Collin Fan, Lavik Jain, Sophie Zhu, Alex Bernat, Kathryn Harper, Jude Partovi, Eli Ocroft, Gianfranco Randazzo

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 $\mathbf{x} = \{x_S\}_{S \subseteq [n]}$.

Algorithm: We define our divide and combine algorithm which takes in a n-subset-dict $\mathbf{x} = \{x_S\}_{S \subseteq [n]}$ and outputs the polar transform $P_n(\mathbf{x})$ as such:

Divide n-subset-dict x into the two halves x_1, x_2 which contain the first $\frac{2^n}{2}$ entries of x and the last $\frac{2^n}{2}$ entries of x, respectively. Thus, let S be the power set of $\{1,...,n\}$, let S_1 be the power set of $\{1,...,n-1\}$ and let $S_2 = S \setminus S_1$. Therefore, $x = \{x_s\}_{s \in S}$, $x_1 = \{x_{s_1}\}_{s_1 \in S_1}$, and $x_2 = \{x_{s_2}\}_{s_2 \in S_2}$.

Recursively solve for the polar transformation $P_{n-1}(x_1)$ on the first half. For x_2, we must first transform before attempting to find the polar transformation of x_2 since x_2 is not a valid n-subset-dict. Let T be a $\frac{N}{2}$ size set indexed by all subsets $p \subseteq P$ where $p = s_2 - \{n\} \in P$ for all $s_2 \in S_2$. Furthermore, let $x_{s_2} = t_p \in T$, again where $p = s_2 - \{n\}$. Then, solve for the polar transformation $P_{n-1}(T)$ on the second half. Finally, define the base case of our recursion when $n = 0$ (and thus $N = 1$), in which case $P_0(x) = x$, where $|x| = 1$.

Combine two subproblems $P_{n-1}(x_1)$ and $P_{n-1}(T)$ using the following definition: $P_n(x) = P_{n-1}(x_1) \bigcup P_{n-1}(x_1) \oplus P_{n-1}(T)$. We define the $\oplus$ operation between two sets as $x \oplus y = \{x_0 \oplus y_0, x_1 \oplus y_1, ..., x_n \oplus y_n\}$ where $x = \{x_0, ..., x_n\}$ and $y = \{y_0, ..., y_n\}$. In addition, note that the union operation $\bigcup$ joins two sets together and preserves the order of the elements of both sets in the union set.

Correctness: We will prove that our algorithm correctly computes the polar transform $P_n(\mathbf{x})$ given n-subset-dict $\mathbf{x} = \{x_S\}_{S \subseteq [n]}$ using induction on n.

Proof. Our base case is when $n = 0$, in which case we have that $S = \{\emptyset\}$. Then, $x = \{x_\emptyset\}$. Since $|x| = 1$, our algorithm then returns $P_0(x) = x = \{x_\emptyset\} = w_\emptyset$ by the definition of the Polar transform (also, see example provided in question). Thus, our base case is proved correct.

In our inductive step, we assume the inductive hypothesis that our algorithm correctly computes $P_{n-1}(x)$ with $x = \{x_S\}_{S \subseteq \{1,...,n-1\}}$. We wish to show that our inductive hypothesis implies our algorithm calculates $P_n(x)$ correctly with $x = \{x_S\}_{S \subseteq \{1,...,n\}}$.

Thus, let S be the power set of $\{1,...,n\}$, let S_1 be the power set of $\{1,...,n-1\}$ and let $S_2 = S \setminus S_1$. Therefore, $x = \{x_s\}_{s \in S}\}$, $x_1 = \{x_{s_1}\}_{s_1 \in S_1}$, and $x_2 = \{x_{s_2}\}_{s_2 \in S_2}$. Thus, it follows $x_1 \bigcup x_2 = x$. Now, let T be a $\frac{N}{2}$ size set indexed by all subsets $p \subseteq P$ where for all $s_2 \in S_2$, $p = s_2 - \{n\} \in P$. Furthermore, let $x_{s_2} = t_p \in T$ where $p = s_2 - \{n\}$. Finally, let $P_{n-1}(x_1) = w_1$ and $P_{n-1}(T) = w_2$, which are both correct polar transformations of x_1 and T by our hypothesis. Let the correct $P_n(x) = w$ and let w' be the half of w which has subsets which do *not* contain $\{n\}$. Similarly, let w'' be the half of w which has subsets which *do* contain $\{n\}$. Thus, we will show that $w = w_1 \bigcup w_1 \oplus w_2$. Since w_2 is the correct polar transformation of T, we know by the definition of polar transformation that for all sets $p \in P$:

$$w_{2_p} = \sum_{Q:Q \subseteq p} t_Q \pmod 2. \tag{6}$$

However, since $t_p = x_{s_2}$ where $s_2 = p + \{n\}$, then for all $s_2 \in S_2$:

$$w_{2_p} = \sum_{Q:Q \subseteq s_2} x_Q \pmod 2. \tag{7}$$

Thus, every w_{2_p} is the sum of all x_Q where $Q : Q \subseteq s_2$. Therefore, since $s_2 = p + \{n\}$, then it follows that all elements of w_2 have at least one of the elements x_i in their sum that contains n in its subset index ($n \in i$). From this fact, it then follows that any element w_{2_k} of w_2 has n in the subset index ($n \in k$), since each x_i which is XORed (or equivalently, summed mod 2) must have i being a subset k. However, none of the elements in w_2 has an element in its sum which contains any x_i where set $i \subset S_1$ by the definition of disjoint sets S_2 and S_1.

In addition, for w_1, we have for all $s_1 in S_1$:

$$w_{1_{s_1}} = \sum_{Q:Q \subseteq s_1} x_Q \pmod 2. \tag{8}$$

Thus, every $w_{1_{s_1}}$ is the sum of all x_Q where $Q : Q \subseteq s_1$. Therefore, since s_1 will never contain n since it is a subset of the power set S_1 for $\{1,...,n-1\}$, then it follows that any element of w_1 does *not* have any element x_i in their sum that contains n in its subset index ($n \notin i$). Furthermore, it follows from similar logic that any element of w_1 is guaranteed to not have n in the subset index (if the element's subset index did contain n, then n would be included in some subset of the powerset of the element's subset index, resulting in a contradiction with the previous statement).

Thus, each entry w_l in $w_1 \oplus w_2$ can be expressed as $w_{2_p} \oplus w_{1_{s_1}}$ (which is also equivalent to $w_{2_p} + w_{1_{s_1}} \pmod 2$, by the problem definition) where $p = s_2 - \{n\} = s_1$ are corresponding entries in w_1 and w_2 as mentioned earlier in our algorithm definition. Furthermore, by the preceding statements we have any $w_l \in w_{2_p} \oplus w_{1_{s_1}} \in w_2 \oplus w_1$ has to contain an element x_i in its sum which contains n in its subset index ($n \in i$). Thus, by our inductive hypothesis, we assert that each $w_l \in w_1 \oplus w_2$ has the correct sum mod 2 of all x_i such that $i \subseteq l$. Finally, from the last 3 paragraphs, we can also

assert that the sums which compose elements in $w_1 \oplus w_2$ the sums of all x_i such that i is a subset of S (the powerset of $\{1,...,n\}$ which also contains $\{n\}$.

Importantly, notice that this definition implies that $w_1 \oplus w_2 = w''$. This is because since S is the powerset of $\{1,...,n\}$, then exactly half of the subsets in S will contain n (since we can either choose to leave n in or out of any subset). Thus, all elements in the other half of the subsets in powerset S must not contain n, so we can represent this other half of the subsets as belonging to the powerset of $\{1,...,n-1\}$, or equivalently the subset w'.

However, recall that S_1 is defined to be the powerset of $\{1,...,n-1\}$, which means that any element in w_1 does not have n in its subset index. Therefore, since in our inductive hypothesis we determined that w_1 and w_2 is calculated correctly, then w' is w_1. Moreover, since we have proved $w_1 = w'$ and $w_1 \oplus w_2 = w''$, it follows that w_1 and $w_1 \oplus w_2$ are disjoint sets that form a set cover for $w = P_n(x)$. Therefore, since our algorithm returns $P_n(x) = w_1 \bigcup w_1 \oplus w_2$, then our algorithm outputs the correct polar transformation $P_n(x)$ given our inductive hypothesis.

Since we have also proved the base case correct, we have proved that out algorithm correctly outputs $P_n(x)$ for any $n \in \mathrm{N}$. □

Runtime: Let s be the number of elements in the n-subset-dict. By definition from the problem description, we have $s = 2^n$. Since we divide each problem into two smaller subproblems and perform $O(s)$ different XOR operations to in our combine step, we can define a a recurrence relation for our algorithm. Under the assumption that taking the XOR of two values is $O(1)$ time, our runtime can be expressed as the recurrence relation $T(s) = 2T(\frac{s}{2}) + O(s)$. Thus, by the Master Theorem, since $a = b = 2$ and $k = 1$, we know that our algorithm has a runtime of $O(s\log(s))$. However, recall that $s = 2^n$, so our algorithm has a runtime of $O(n2^n)$.

(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 $\mathbf{x} = \{x_S\}_{S\subseteq[n]}$ for which $\mathbf{w} = P_n(\mathbf{x})$.

Algorithm: We define our algorithm to *invert* the polar transform which takes an n-subset-dict $\mathbf{w} = \{w_S\}_{S\subseteq[n]}$ and computes $\mathbf{x} = \{x_S\}_{S\subseteq[n]}$ for which $\mathbf{w} = P_n(\mathbf{x})$ as such:

Let $P_n(x)$ be the algorithm for computing the polar transformation which we found and proved in part (a). Return $P_n(w)$.

Correctness: We will prove that our algorithm correctly inverts the polar transform.

Proof. Let S be the power set of $\{1,...,n\}$. By the definition of the polar transformation,

we know that any w_s, where $s \in S$, is defined as:

$$w_s = \sum_{T:T\subseteq s} x_T \pmod 2. \tag{9}$$

Therefore, we can use this definition and apply the principle of inclusion-exclusion to isolate for x_s.

$$x_s = \sum_{T:T\subseteq s} x_T \pmod 2 - \sum_{s\setminus\{e_i\}} \sum_{T:T\subseteq s\setminus\{e_i\}} x_T \pmod 2 + \sum_{s\setminus\{e_i,e_j\}} \sum_{T:T\subseteq s\setminus\{e_i,e_j\}} x_T \pmod 2$$

$$- \sum_{s\setminus\{e_i,e_j,e_k\}} \sum_{T:T\subseteq s\setminus\{e_i,e_j,e_k\}} x_T \pmod 2 + \ldots + (-1)^n \sum_{\emptyset} \sum_{T:T\subseteq\emptyset} x_T \pmod 2$$

Where e_i, e_j, e_k... are arbitrary elements in s. Therefore, we assert $s \setminus \{e_i, e_j, e_k\}$ is the set of all possible subsets of s with three arbitrary elements removed ($|s \setminus \{e_i, e_j, e_k\}| = |s| - 3$. Then, equivalently, if we substitute, we have:

$$x_s = (w_s - \sum_{s\setminus\{e_i\}} w_{s\setminus\{e_i\}} + \sum_{s\setminus\{e_i,e_j\}} w_{s\setminus\{e_i,e_j\}} - \sum_{s\setminus\{e_i,e_j,e_k\}} w_{s\setminus\{e_i,e_j,e_k\}} + \ldots + (-1)^n w_\emptyset) \pmod 2$$

However, since we have identity $-1 \pmod 2 = 1 \pmod 2$, our summations are equivalent to:

$$x_s = (w_s + \sum_{s\setminus\{e_i\}} w_{s\setminus\{e_i\}} + \sum_{s\setminus\{e_i,e_j\}} w_{s\setminus\{e_i,e_j\}} + \sum_{s\setminus\{e_i,e_j,e_k\}} w_{s\setminus\{e_i,e_j,e_k\}} + \ldots + w_\emptyset) \pmod 2$$

Importantly, note that this definition is equivalent to:

$$x_s = \sum_{T:T\subseteq s} w_T \pmod 2$$

Recall that this is the definition of a polar transform of $\mathbf{w} = \{\mathbf{w_S}\}_{\mathbf{S}\subseteq[\mathbf{n}]}$, since we previously defined that $s \subset S$, where S is the powerset of $\{1, ..., n\}$. Thus, it follows that the polar transform $\mathbf{w} = P_n(\mathbf{x})$ and $\mathbf{z} = P_n(\mathbf{w})$. Therefore, to invert the polar transform $\mathbf{w} = P_n(\mathbf{x})$ and get $\mathbf{x} = \{x_S\}_{S\subseteq[n]}$, we calculate $P_n(\mathbf{w})$. Thus, by the correctness of our algorithm from part(a), our algorithm correctly inverts the polar transform. □

Runtime:

Proof. Let s be the number of elements in the n-subset-dict $\mathbf{w} = \{w_S\}_{S\subseteq[n]}$. By the problem description, we then have $s = 2^n$. Since the only step in our algorithm is calling our algorithm from a on $\mathbf{w}$, then we know from our runtime proof in part (a) that our algorithm has a runtime of $O(s \log s)$ which is $O(n2^n)$. □

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 $1 \le \ell \le d$, 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)

Algorithm: We will define our algorithm which takes in a list of points $L = \{\mathbf{z}_1, \ldots, \mathbf{z}_n\}$ in two dimensions and 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 as such:

Create array A of length n indexed by the points in L such that for any element $a_i \in A$, a_i is the number of points that z_i covers. Since a point by definition covers itself, set all $a_i \in A$ to 1. First, sort L by x-coordinate using merge sort. Assume for simplicity that n is a power of 2.

Perform a modified merge sort on the sorted L, this time considering the sorting with regard to the y-coordinate of each point:

Recursively divide the x-sorted L in half until we have n sublists of length 1. Then, when we combine/merge two sublists together, we initialize a counter c and a merge list to store the merged sublists. We define two pointers which start at the beginning of the two sublists which we merge. Then, we compare the element in each sublist at each pointer and add the element with the smaller y-coordinate of the two, advancing the pointer of the sublist whose element was added to the merge list. For each element which is added to the merge list from the left sublist, increment c by 1. For each element z_i which is added to the merge list from the right sublist, add the current value of c to a_i in the array A. If two points being compared have the same y-coordinate, add the point with a smaller x-coordinate first (add the point from the left sublist first). Note that this is essentially merge sort, with the only modifications being the counter c and the extra checks for which sublist a point was added from.

Finally, once all merges have been completed, return A.

Correctness: We will prove that our algorithm outputs the correct 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.

Proof. To prove the correctness of our algorithm, we will prove inductively that after any merge of size $\frac{n}{2}$ size sublists in our algorithm, any point z_i in the resulting n size merged list has a_i accurately accurately reflecting the number of points in the merged list which z_i covers.

We will use the invariant that at any merge in our algorithm, the elements in the right sublist of the merge have x-coordinates which are larger than the elements in the left sublist of the merge. This is proven by the correctness of merge sort: since we first sort the list L by x-coordinate, we know that no point z_j which comes before another point z_i in the x-sorted L can cover z_i because its x-coordinate must be smaller than

z_i.
Our base case is when $n = 1$, when our list is size 1. Recall that we set all $a_i = 1 \in A$ initially. Since each point by definition covers itself and there is only 1 point in a sublist of size 1, every point in its sublist of length 1 has a correct a_i in A. Our base case is proved true.

In our inductive step, we assume, for simplicity, that for any sublist of size n, our algorithm correctly updates a_i such that any point z_i in the n size sublist has a_i accurately accurately reflecting the number of points in the sublist which z_i covers. We will prove that given this hypothesis, the resulting size $2n$ merged array merge correctly maintains the array A. Let us define sublists l and r of size n which have been correctly updated by our algorithm, such that l is the left sublist in the merge and r is the right sublist in the merge. By our invariant, we know that in any merge, any point in l cannot cover a point in the r. Furthermore, since our algorithm combines the r and l by comparing the y-coordinates of the points at each "pointer" for r and l, all points in the merged list $z_j ... z_k$ which come before a point z_i will have a y-coordinate less than z_i (this is based on the correctness of mergesort as well). Now consider if a point z_j is added to the merged list from l while a point z_i is added *after* z_j to the merged list from r. By our invariant, the x-coordinate of z_i must be greater than z_j. Furthermore, by the correctness of the combine step in merge sort, we know that the y-coordinate of z_i must be greater than z_j. It then follows that z_i covers z_j by definition. Therefore, since we increment the counter for each element added to the merged list from l and increment a_i by the current value of c when we add z_i from r, our algorithm correctly maintains the values of $a_i \in A$ when merging 2 size n sublists (assuming our inductive hypothesis).

Thus, we have proved by induction that after any merge of size $\frac{n}{2}$ size sublists in our algorithm, any point z_i in the resulting n size merged list has a_i accurately accurately reflecting the number of points in the merged list which z_i covers. Therefore, after the last merge in our algorithm in which we merge the two halfs of L back together, all points z_i will have the correct a_i. □

Runtime:

Proof. Let n be the number of points in L. We operate under the assumption that incrementing the counter and updating values of a_i in A take $O(1)$ time. Since we sort all points by their x-coordinate at first using merge sort, we know by the runtime of merge sort that this step takes $O(n \log n)$ time. Then, our algorithm performs a slightly modified merge sort using the y-coordinates of each point. Since we recursively split the list in two halves and for each combine step we perform $O(n)$ comparisons, increments, and updates (which each take constant O(1) time), then our runtime for this step can be expressed as the recurrence relation $T(n) = 2T(\frac{n}{2}) + O(n)$. By the Master Theorem, our runtime for this step therefore is $O(n \log n)$. Thus, our total runtime for our algorithm is $O(n \log n + n \log n)$ which is $O(n \log n)$. □

(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.

Algorithm: We define our algorithm as follows:

i. **Define** ComputePenalty(A):

- If $A \geq 0$, then return A^3.
- Else, return $2^{-A} - A^3 - 1$.

ii. **Define** PrintNeatly($words$, M):

- Initialize n to the number of words.
- Create an array D of length $n+1$ and initialize all elements to ∞. Set $D[-1]$ to 0 (last line has no penalty).
- Create an array B of length n and initialize all elements to 0.
- **For** each i from 0 to $n-1$:
  - Set l to 0.
  - **For** each j from i down to 0:
    - Increment l by the length of words[j].
    - Calculate A as $M - l - i + j$.
    - Calculate the penalty P using ComputePenalty(A).
    - If i is $n-1$ and $A \geq 0$, set P to 0.
    - If $D[j-1] + P < D[i]$, update $D[i]$ to $D[j-1] + P$ and set $B[i]$ to $j-1$.
- Initialize *lines* as an empty string.
- Starting with k equal to $n-1$, **while** $k \geq 0$:
  - Join words from $B[k]+1$ to k, inclusive, with spaces to form a *line*.
  - Prepend *line* and a newline character to *lines*.
  - Set $k = B[k]$.
- Return $D[n-1]$ and *lines*.

We will describe the intuition behind our algorithm. We define a helper function ComputePenalty(A) such that it returns A^3 if $A \geq 0$ and $2^{-A} - A^3 - 1$ if $A < 0$. Let our input be a set of n words with lengths $\ell, ..., \ell_{n-1}$. We define a DP array D of length $n+1$ such that for each $D[i]$, where $0 \leq i \leq n-1$, we define $D[i]$ to be the lowest penalty achieved for the paragraph containing the first i words with lengths $\ell_0, ..., \ell_i$. We also define $D[-1] = 0$ and $D[1, ..., n-1] = \infty$. Our recursive definition of the optimal value is thus:

$$D[i] = \min_{0 \leq j \leq i} (D[j-1] + P_{\{j,i\}})$$

Where $P_{\{j,i\}}$ denotes the penalty associated with adding all words j through i onto the same line. Therefore, $P_{\{j,i\}}$ = ComputePenalty(A) where $A = A = M - j + i - \sum_{k=j}^{i} \ell_k$. Importantly, when computing $D[i] = D[j-1] + P_{\{j,i\}}$, we check if $i = n-1$ and $A \geq 0$, in which case $P_{\{j,i\}} = 0$. Finally, let us define an array B of length n such that for any $B[i] = t$, t is the index for at which we add the last line break when calculating $D[i]$.

Thus, we begin our algorithm by iterating through $0 \leq i \leq n-1$. For each i, we iterate backwards from $i \geq j \geq 0$, calculating $D[j-1] + P_{\{j+1,i\}}$ as defined above. For each j, we compare the value of $D[j-1] + P_{\{j+1,i\}}$ to the current stored value of $D[i]$. If for any j, $D[j-1] + P_{\{j+1,i\}} < D[i]$, set $D[i] = D[j-1] + P_{\{j+1,i\}}$. As mentioned before, when we calculate the cost of $D[j-1] + P_{\{j+1,i\}}$, we check if $i = n-1$ and $A \geq 0$, in which case we set $P_{\{j+1,i\}} = 0$. At the end, once we have iterated through all i, we return $D[n-1]$ which is the minimum penalty incurred over all lines in the neatest way to print the paragraph. We also define $k = n-1$, and while $k \geq 0$, prepend the words from index $B[k]+1$ to k, inclusive, with spaces between the words as a new line in the paragraph. Then, return the paragraph as well.

The pseudocode for our algorithm is illustrated above.

Correctness: We will prove that our algorithm finds the neatest way to print a paragraph, or equivalently, our algorithm finds the formatting of words into lines such that the penalty over all lines is minimized.

Proof. To prove our algorithm's correctness, we will first prove that any $D[i]$, where $0 \leq i \leq n-1$ represents the lowest penalty possible for a paragraph consisting of words 0 through i with associated lengths $\ell_0 ... \ell_i$. We will prove this using strong induction.

Our base case is when $i = 0$. Since when $i = 0$, we consider the case where our paragraph only consists of one word with associated length ℓ_0. Thus, there is only one possible configuration of lines in the paragraph. Furthermore, the associated cost, by definition of the problem, is the penalty of a line containing just word 0 with recommended line length M. By the definition of $P_{\{j,i\}}$, this is equivalent to $P_{\{0,0\}}$, or ComputePenalty(A) where $A = M - 0 + 0\ell_0$. Thus, our algorithm computes $D[0] = \min_{0 \leq j \leq 0}(D[j-1] + P_{\{j,i\}}) = D[-1] + P_{\{0,0\}} = 0 + P_{\{0,0\}} = P_{\{0,0\}}$. Our base case is therefore correct.

In our inductive step, we assume the inductive hypothesis that $D[0], D[1], D[2], ..., D[i]$ is the correct minimum penalty for paragraphs consisting of words 0 through words i. We wish to show that given this hypothesis, $D[i+1]$ is the correct minimum penalty for words 0 through words $i+1$. Importantly, note that we cannot naively set $D[i+1] = D[i] + P_{\{i+1,i+1\}}$ since $P_{\{i+1,i+1\}}$ is not guaranteed to be the minimum penalty possible (because we are penalized if the length of a line is over OR under the optimal). Thus, let us consider the problem of adding the last line break in the paragraph consisting of words 0 through words $i+1$. Let j be the index of the word which begins the last line in our subproblem paragraph. Thus, words j through words $i+1$ are on the last line of this paragraph by definition. However, it follows that the rest of the paragraph consists of words 0 through words $j-1$. In addition, the total penalty of our subproblem $i+1$ can be framed as the penalty for adding words j through words $i+1$ onto a line and the penalty of a paragraph consisting of words 0 through words $j-1$. Thus, the minimum penalty for our subproblem $D[i+1]$ is the value of j which minimizes the sum outlined in the previous statement. Then, if we consider all possible starting points to our last line (all possible word j, for $0 \leq k \leq i+1$), we are guaranteed to find the minimum penalty for the paragraph consisting of words 0 through words $i+1$. However, it is essential to note that optimizing the total penalty of the paragraph, or equivalently the penalty for adding words j through words $i+1$ onto a line and the penalty of a paragraph consisting of words 0 through words $j-1$, means we must find the optimal penalty of a paragraph consisting of words 0 through words $j-1$. According to our inductive hypothesis, this optimal penalty is equivalent to $D[j-1]$. Therefore, the total penalty of the paragraph can be expressed as $D[i+1] = \min_{0 \leq j \leq i+1}(D[j-1] + P_{\{j,i+1\}})$. By our algorithm construction, it is clear we iterate backwards $j = i+1$ to $j = 0$, calculating $P_{\{j,i+1\}}$ each time and adding it to $D[j-1]$. We also store the minimum of the sum, so that at the end of the iterations of j, we have found the minimum $D[j-1] + P_{\{j,i+1\}}$ and set $D[i+1] = D[j-1] + P_{\{j,i+1\}}$. Thus, our algorithm correctly implements the strategy we outlined previously. Therefore, given the inductive hypothesis, our algorithm calculates the correct minimum penalty for a paragraph containing words 0 through words i+1 and stores it in $D[i+1]$. Finally, based on the problem description, we must consider a case where we add the last line to the paragraph containing all $n-1$ words. In this case, if $A \geq 0$, where $A = M - j + i - \sum_{k=j}^{i} \ell_k$, the penalty of the last line is 0. By the definition of our subproblem construction, this is when $i = n-1$ (where we consider words 0 through words n-1). Since we check if $i = n-1$ and $A \geq 0$ in every calculation of $D[j-1] + P_{\{j,i\}}$ and set the penalty $P_{\{j,i\}} = 0$ if the conditions are satisfied, then our algorithm maintains its correctness in the edge case $i = n-1$ (because we have previously defined $P_{\{j,n-1\}}$ as the penalty adding the words j through the last word in the paragraph onto a line).

Therefore, since we have proved the base case and $D[0] \wedge D[1] \wedge ... \wedge D[i]$ implies the correctness of $D[i+1]$, we have proved that for any $0 \leq i \leq n-1$, D[i] is the optimal minimum penalty for a paragraph containing words 0 through words i. It follows that $D[n-1]$ is the optimal minimum penalty for a paragraph containing all n

words with associated lengths $\ell_0, ..., \ell_n$.

Now, we will prove that our algorithm outputs the neatest way to print a paragraph (since the question does not solely ask for the minimum penalty). We have previously shown that $D[n-1]$ is the optimal minimum penalty for a n word paragraph. In addition, in our induction proof, we have shown that for each $D[i]$, we find an index of a word j which starts the last line of that i word paragraph. In addition, by the definition of our algorithm, we store index $j-1$ in $B[i]$ (the same i for D[i] and B[i]). Thus, for D[n-1], B[n-1] stores the index of the last word in the second to last line of the entire paragraph. Generally, for any D[i], B[i] stores the index of the last word in the second to last line of the paragraph containing the words from word 0 to word i. Therefore, when we form lines by starting at $k = n-1$ and adding word $B[n-1]+1$ to word k (including spaces, inclusive), we are essentially prepending the last line of the n word optimal paragraph. Then, when we set $k = B[k]$ and then repeat this process, it is clear that when our while loop terminates, we have outputted the neatest way to print the paragraph. □

Runtime:

Proof. Let n be the number of words in the paragraph. Since we build our subproblems from $0 \leq i \leq n$, we can see that we have $O(n)$ subproblems. For each subproblem (outer loop of algorithm), we iterate backwards from i to 0 to calculate the minimum penalty for that subproblem. Hence, each subproblem takes $O(n)$ computations of $D[j-1] + P_{\{j,i\}}$. Then, assuming that the arithmetic operations involved in calculating $P_{\{j,i\}}$ takes O(1) time and indexing D takes O(1), our total runtime of the algorithm will be $O(n * n)$ which is $O(n^2)$.

□

Space Complexity:

Proof. In our algorithm, we have two dynamic programming arrays B and D. B has size n by definition, and D has size $n+1$ by definition. Thus, our algorithm has a space complexity of $O(n) + O(n+1)$ which is $O(n)$. □

(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.)

M = 40: The minimum penalty is 396. The corresponding optimal division of words into lines is:

> 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.)

M = 72: The minimum penalty is 99. The corresponding optimal division of words into lines is:

> 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 $x_1 \le \cdots \le x_n$. 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 proba-

bility 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 $p_1, \ldots, p_n, q_0, \ldots, q_n$.

Algorithm: Let us define our algorithm that constructs a binary search tree that minimizes the expected number of internal nodes queried as such:

Let $D[i,j]$ be a 2-dimensional DP array defined such that $D[i,j]$ is the minimum expected number of internal nodes queried from a binary search tree including the books $x_i, ..., x_j$ and corresponding intervals $q_{i-1}, q_i, ..., q_j$. Let $R[i,j]$ be a 2-dimensional DP array defined such that $R[i,j] = x_r$, where x_r is the root of the optimal binary search tree including the books $x_i, ..., x_j$ and corresponding intervals $q_{i-1}, q_i, ..., q_j$ (we define "optimal" to mean the BST that minimizes the expected number of internal nodes queried). Finally, we define $P[i,j]$ to be a 2-dimensional DP array defined such that $P[i,j] = \sum_{n=i}^{j} p_n + \sum_{m=i-1}^{j} q_m$.

We define our recursive definition of the minimum value as such:

$$D[i,j] = \min(C_1(i,j), C_2(i,j), C_3(i,j)$$

$$D[i,i] = 1$$

$$C_1(i,j) = \frac{p_i}{P[i,j]} + \frac{q_{i-1}}{P[i,j]} + (D[i+1,j]+1)\frac{P[i+1,j]}{P[i,j]}$$

$$C_2(i,j) = \min_{i<r<j} \left(\frac{p_r}{P[i,j]} + (D[i,r-1]+1)\frac{P[i,r-1]}{P[i,j]}\right) + (D[r+1,j]+1)\frac{P[r+1,j]}{P[i,j]}\right)$$

$$C_3(i,j) = \frac{p_j}{P[i,j]} + \frac{q_j}{P[i,j]} + (D[i+1,j]+1)\frac{P[i,j-1]}{P[i,j]}$$

Additionally, let $R[i,j] = x_r$, where $r = i$ if $C_1(i,j)$ is the minimum, $r = j$ if $C_3(i,j)$ is the minimum, or r is the value of r that minimizes $C_2(i,j)$ if $C_2(i,j)$ is the minimum of the three cases. For all $R[i,i]$, set $R[i,i] = x_i$.

Thus, let us define our algorithm as such:

- Set all $D[n,n] = 1$ for $1 \leq n \leq n$
- Initialize $P[i,j]$ as defined above
- For $1 \leq l \leq n-1$ (the length of the interval i to j we consider)
  - For $1 \leq i \leq n-l$
    - $j = i + l$
    - $D[i,j] = \min(C_1, C_2, C_3)$
    - $R[i,j]$ calculated and stored as outlined above
- Return $D[1,n]$
- To reconstruct the optimal BST:
  - Set $x_r = R[i,j]$
  - Recursively find $x_r = R[i,r-1]$ and $x_r = R[r+1,j]$ until $x_r = R[n,n] = x_r$
  - Insert intervals I_j between the last level of internal nodes

Correctness: We will prove that our algorithm outputs the BST which minimizes the expected number of internal nodes queried.

Proof. To prove the correctness, we will prove that $D[i,j]$ is the minimum expected value of internal nodes queried for a BST including books $x_i, ..., x_j$ and intervals $q_{i-1}, q_i, ..., q_j$. We will prove this with strong induction on the length l of the interval i to j.

Our base case is when $l = 0$, or equivalently, $i = j$. It follows from the problem definition and our definition of D that our BST consists of exactly one book x_i as the root of the BST and two intervals I_{i-1} and I_i. Thus, since x_i is the root, every query must query x_i. On the other hand, since x_i is the only root in the BST, every query at most queries through 1 internal node. This means that the expected number of queries through internal nodes when $l = 0$ is exactly 1. Hence, since we defined $D[i,i] = 1$, our base case is proved correct.

In our inductive step, we assume that for every $D[i,j]$ such that $0 \leq l = j - i \leq s$, $D[i,j]$ is the correct minimum expected value of internal nodes queried for a BST including books $x_i, ..., x_j$ and intervals $q_{i-1}, q_i, ..., q_j$. We will show that given this inductive hypothesis, every $D[i,j]$ with $l = j - i = s + 1$ is correct minimum expected value of internal nodes queried for its corresponding BST outlined above. We will prove each case C_1, C_2, C_3 from above.

For case C_1, we essentially consider if x_i is the root of the BST. Thus, our BST will contain the nodes $x_{i+1}, ..., x_j$ to the right of root x_i by definition of a BST, and the interval probability q_{i-1} as the left child of x_i. Thus, by the definition of expected value, we consider the number of internal nodes queried when we go down the BST to the left, when we go down the BST to the right, and when we stop our query at the root x_i. Since the left only contains q_i and to get to q_i we must first pass through the root, it is clear that the expected

number of internal nodes queried is $\frac{q_i}{P[i,j]} * 1$ (the normalized probability of reaching node q_i multiplied by the number of internal nodes queried. This logic will be used in all following explanations). When we stop our query at the root x_i, we still query one internal node x_i so our expected number of internal nodes queries is $\frac{x_i}{P[i,j]} * 1$. Finally, when we consider a query which goes through the right side of the BST, we consider a subtree with internal nodes $x_{i+1}, ..., x_j$. However, if $j - i = s + 1$, then $j - (i + 1) = s$. Thus, this subtree $x_{i+1}, ..., x_j$ has a length $l = s$. Thus, by our inductive hypothesis, the expected value of internal nodes queried is $D[i + 1, j]$. Hence, the expected number of internal nodes queried when going down the BST from the right of x_i is $(D[i+1, j]+1)\frac{P[i+1,j]}{P[i,j]}$ ($D[i+1, j]+1$ because going down the right requires us to first query through x_i, so we must add one to the expected number of internal nodes queried). Therefore, our expected number of internal nodes queried for C_1 when our root is x_i is the sum of these cases, which is $\frac{p_i}{P[i,j]} + \frac{q_{i-1}}{P[i,j]} + (D[i+1, j]+1)\frac{P[i+1,j]}{P[i,j]}$.

Similarly, for C_3, we essentially consider if x_j is the root of the BST. By similar logic to C_1, the BST will contain the nodes $x_i, ..., x_{j-1}$ to the left of root x_i and the interval probability q_j as the left child of x_j. Furthermore, it follows from the logic in C_1 that the expected number of internal nodes queried when we go down the right of the BST or when we stop our query at x_j is $\frac{p_j}{P[i,j]}$ and $\frac{q_j}{P[i,j]}$, respectively. Finally, since the left side of the BST contains nodes $x_i, ..., x_{j-1}$, we again deal with a subtree with $l = s$. Therefore, by the logic in case C_1, the expected number of internal nodes queried when going down the left of the BST is $(D[i + 1, j] + 1)\frac{P[i,j-1]}{P[i,j]}$. Therefore, the expected number of internal nodes queried for case C_3 when our root is x_j is the sum $\frac{p_j}{P[i,j]} + \frac{q_j}{P[i,j]} + (D[i+1, j]+1)\frac{P[i,j-1]}{P[i,j]}$.

Finally, for C_2, we consider all cases where the root of the BST is not j or i. Let x_r denote a possible root of the BST such that $i < r < j$. Thus, using the logic we outlined in C_1 and C_3, the expected number of internal nodes queried when we stop our query at x_r is $\frac{p_r}{P[i,j]}$. When our query goes to the left side of the BST, we know by the definition of the BST that this half of the tree contains the internal nodes $x_i, ..., x_{r-1}$. However, if $i < r < j$ and $j - i = s + 1$, then $r - 1 - i < s$. Then, by our inductive hypothesis, the expected number of internal nodes is $D[i, r - 1]$ and the expected number of internal nodes queried when going down the left side of the BST is $(D[i, r - 1] + 1)\frac{P[i,r-1]}{P[i,j]})$. By similar logic, the expected number of internal nodes when querying down the right side of the BST is $D[r + 1, j]$ and the expected number of internal nodes queried when going down the left side of the BST is $(D[r+1, j]+1)\frac{P[r+1,j]}{P[i,j]})$. Therefore, the expected number of internal nodes queried for a given x_r where $i < r < j$ is $(\frac{p_r}{P[i,j]} + (D[i, r - 1] + 1)\frac{P[i,r-1]}{P[i,j]}) + (D[r + 1, j] + 1)\frac{P[r+1,j]}{P[i,j]})$. Furthermore, since we find the minimum value of $(\frac{p_r}{P[i,j]} + (D[i, r - 1] + 1)\frac{P[i,r-1]}{P[i,j]}) + (D[r + 1, j] + 1)\frac{P[r+1,j]}{P[i,j]})$ over any $i < r < j$, it follows that C_2 is the minimum expected number of internal nodes queried for any starting root x_r where $i < r < j$ in our BST.

We know C_1, C_2, and C_3 cover all possible starting roots of our BST (which is determinstic given the root). Therefore, since we define $D[i, j] = \min(C_1, C_2, C_3)$, it follows that $D[i, j]$ is the minimum expected number of internal nodes queried when $l = j - i = s + 1$, assuming our inductive hypothesis. As we have proved the base case as well, we have by

induction that $D[i, j]$ is the minimum expected value of internal nodes queried for a BST including books $x_i, ..., x_j$ and intervals $q_{i-1}, q_i, ..., q_j$.

Now, we will show that our reconstruction of the optimal BST is correct. Since we defined $R[i, j]$ to be root of the optimal BST containing books $x_i, ..., x_j$ and intervals $q_{i-1}, q_i, ..., q_j$, it follows that $R[1, n]$ is the root of the optimal subtree containing all books and intervals. Then, by definition of a BST, the left and right side of this root consist of two BST's containing $x_1, ..., x_{r-1}$ and $x_{r+1}, ..., x_j$, respectively. Therefore, to find the two children of the root stored in $R[1, n]$, we recursively find $R[1, r-1]$ and $R[r+1, j]$ to find the root of the optimal sub-BST's outlined in the previous statement. Thus, after repeating this process, we can see that our algorithm correctly reconstructs the optimal BST. □

Runtime:

Proof. Let n be the number of books in inventory. Therefore, since our algorithm has three nested loops (for every $l = j - i$, we loop through all possible $1 \leq i \leq n - l$ and for every i, we loop through all $i \leq r \leq j = i + l$ to calculate $D[i, j]$). Therefore, this step in our algorithm takes $O(n^3)$ time. When we initalize $P[i, j]$, we must perform two nested loops to capture all pairs (i, j). Thus, this step takes $O(n^2)$ time. Thus, our algorithm has a total runtime of $O(n^3 + n^2)$ which is $O(n^3)$. □

Space Complexity:

Proof. Since we create 3 2-dimensional arrays in our algorithm, our algorithm's space complexity is $O(n^2)$. □