CS 124 Homework 4: Spring 2024

Collaborators:

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

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

Assume input keys are ordered as shown in the problem definition. If they aren't, sort the input dict so that set keys are in increasing order according to the bit representation of their index set. For example, when $n = 3$, this ensures the ordering $[\{\}, \{1\}, \{2\}, \{1,2\}, \{3\}, \{1,3\}, \{2,3\}, \{1,2,3\}]$ identified by bit strings $[0, 1, 10, 11, 100, 101, 110, 111]$. Note that this bit ordering places each new element a power of two apart from its predecessor: $\{\} = 0, \{1\} = 1, \{2\} = 10, \{3\} = 100, \{4\} = 1000$, etc.

Run this algorithm after preprocessing to compute the polar transform: Define a recursive function `make_polar` that accepts an n-subset dict `nsub` and returns the polar transformation. If `nsub` has one entry, just return that. Else, split `nsub` into equal left and right halves. This will always be an even partition because the size of `nsub` is a power of two. Recursively call `make_polar` on both halves. Instantiate a return vector called `combined` and fill it with the left result. Then, for every corresponding pair `left[i]` and `right[i]`, push `left[i]` $\oplus$ `right[i]` to `combined`. Finally, return `combined`. Pseudocode:

```
fn preprocess(nsub: Vec(Bit)):
    sort nsub in increasing order by bit representation
    of the index set

fn make_polar(nsub: Vec(Bit)) -> Vec(Bit):
    if nsub.len < 2:
        return nsub;

    let partition = nsub.len / 2;
    let left = make_polar(nsub[0..partition]);
    let right = make_polar(nsub[partition..nsub.len]);

    let combined = left.clone();
    for ind in 0..left.len:
        combined.push(left[ind] ^ right[ind]);
    return combined;
```

Correctness:
Prove by induction that the algorithm computes a polar transform of a sorted n-subset dict.

Base case 1: When $n = 0$, an input map of $[x_1]$ is returned as $[w_1] = [x_1]$. This correctly follows from the definition of a polar transform. Using the example above, this is equivalent to $P(\{\}) = \{\}$.

Base case 2: When $n = 1$, an input map $[x_1, x_2]$ is returned as $[x_1, x_1 \oplus x_2]$, which also correctly follows the definition of a polar transformation.

Inductive hypothesis: Consider the case $n+1$. By definition, there are 2^{n+1} distinct entries in the input dict. Dividing it in half yields two sub-dicts of size $\frac{2^{n+1}}{2} = 2^n$. Call `make_polar` on the left and right sub-dicts. By the inductive hypothesis, both return proper polar transformations of their input vectors.

The `combined` dict begins with every entry from the left side. This ensures every polar dict entry with n or fewer members from 0 to n is returned.

Next, assume the keys are ordered as explained in the preprocessing step. Before the recursive call, the input dict is split into equal left and right halves, both of size 2^n. Because there are exactly 2^{n+1} entries in the input map, every entry on the left half has a bit representation from all 0s to n consecutive 1s. The first entry on the right half has a 1 in the $n+1$th bit followed by all 0s and counts upwards to a final entry of $n+1$ consecutive 1s. The returned left array represents the correct polar transformation of every set indexed by the bit strings not including the $n+1$th bit. Because the only difference in the bit string set keys on the left and right dicts is a 1 at the $n+1$th bit position, computing an xor between corresponding left and right dict entries is equivalent to computing the polar transform for every possible set containing the new value encoded at the $n+1$th bit position.

Aside: Purely for example (not rigorous), consider the $n = 2$ case. At $n+1$, there are two subarrays of size $2^2 = 4$. Here, the left array holds entries $[0 = \{\}, 1 = \{1\}, 10 = \{2\}, 11 = \{2,1\}]$, and the right array holds entries $[100 = \{3\}, 101 = \{3,1\}, 110 = \{3,2\}, 111 = \{3,2,1\}]$. When a 1 bit represents the presence of a set member, notice that corresponding entries in the left and right differ only by the leftmost bit.

Thus, pushing `left[ind]` $\oplus$ `right[ind]` for all indices in the left and right dicts is equivalent to pushing the polar transform of every set that includes the $n+1$th element. The combined output is a dict of every polar-transformed entry from positions 0 to $2^n - 1$ and 2^n to $2^{n+1} - 1$. By returning this complete polar transformation, the inductive step is proven.

So, given any input n-subset dict, the dict is first sorted as a preprocessing step. Then, `make_polar` is called. By the inductive proof above, calling `make_polar` on this input yields a complete polar transformation of the input. Thus, given valid input, the algorithm returns correct output.

The only thing returned by the algorithm is the result of `make_polar`, which, by the inductive proof above, is a valid polar transform of the input dict. Thus, if anything is returned, it's a valid polar dict. By these two cases, the algorithm must be correct.

Runtime:
Let n be defined in the context of an n-subset dict such that there are $k = 2^n$ entries

in an input n-subset dict. If the initial dict is not sorted, it is sorted in the preprocessing step. This has a one-time cost of $\Theta(k \log k)$.

The code for `make_polar` is a recurrence. Given input of size k, each frame makes two calls to `make_polar` with input of size $\frac{k}{2}$ and performs $\Theta(k)$ work to construct the output. This yields the recurrence $T(k) = 2T(\frac{k}{2}) + \Theta(k)$. Under a Master Theorem representation where $a = 2$, $b = 2$, and $f(k) = \Theta(k)$, Case II applies because $k \log^0 k = \Theta(k)$ when $k = 0$. This yields a recurrence runtime of $\Theta(k \log k)$. Because this is the same as the preprocessing step, the total runtime must be $\Theta(k \log k)$.

(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 $\mathbf{w} = P_n(\mathbf{x})$.

Consider this algorithm: Given an input polar-transformed dict, call the preprocess function from part (a) to guarantee the bit ordering of the dict keys. Then, call `make_polar` and return its result. This algorithm relies completely on the assertion that polar transformations invert themselves. This is proven below and arises from the observation of how xor behaves under the parallel structure of divided subproblems.

Correctness:
Prove that polar transformations invert themselves. Speficially, that given a polar dict P, that `make_polar(P)` produces a dict D such that `make_polar(D)` is equal to P. Prove by induction on n:

Base case 1: When $n = 0$, the input map $[w_0]$ is returned as $[w_0]$. Because the polar transform of $[x_0]$ is equal to $[x_0]$, it must be the case that $w_0 = x_0$, and this is a correct inversion of the polar transformation.

Base case 2: When $n = 1$, the input map $[w_0, w_1]$ is returned as $[w_0, w_0 \oplus w_1]$. Because this expands to $[x_0, x_0 \oplus x_0 \oplus x_1] = [x_0, x_1]$, this is a proper inversion of the polar transform.

Inductive hypothesis: Assume for $1 \leq n$, `make_polar` correctly inverts a polar dict of entries w_0 to w_{k-1} into the input dict of entries x_0 to x_{k-1} such that calling `make_polar` on the x dict generates the w dict.

Inductive step: Consider a polar dict of size 2^{n+1} with entries w_0 to w_{k-1}. Again assume the entries are ordered as described at the beginning of part (a). Divide the entries in half such that the left half contains entries w_0 to w_{2^n-1} and the right half contains entries w_{2^n} to $w_{2^{n+1}-1}$. Call `make_polar` on both halves. By the inductive hypothesis, this returns a valid inversion of the polar transform. Now, every inverted value in the left half represents an entry in the output dict x_0 to x_{2^n-1}. Assert from part (a) the structure of the bit-index ordering such that corresponding indices in the left and right halves differ only in that the $n+1$th bit on every entry on the right side is 1 while the corresponding bit is 0 on every entry in the left side. Again, this implies that the corresponding entries only differ by a single value, which is the largest element in the right entry's set. By computing the xor of corresponding left and right-side entries and then adding them to the result array, the inverted values x_{2^n} to $x_{2^{n+1}-1}$ are produced. Copying over the entries x_0 to x_{2^n-1} from the left side yields

a complete inversion of the given polar dictionary at $n+1$. This proves the inductive step.

By part (a), assert the correctness of `make_polar`. By the inductive proof above, assert that calling `make_polar` on the given polar dictionary will produce an n-subset dict that, when processed through `make_polar` again, produces the input dict. Thus, if the algorithm returns anything, it returns a valid inversion of the polar transformation.

Similarly, on the given input, if an inversion of the polar transformation exists, `make_polar` will compute it such that the polar transformation of the polar transformation is the input. Thus, if an inversion of the given polar transformation exists, it is returned.

Runtime:
The runtime of this algorithm is the same as part (a). Input is sorted during the preprocessing step, and a single call to `make_polar` on a dict of side k takes $\Theta(k \log k)$ time. Together, these produce an inversion runtime of $\Theta(k \log k)$.

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 $1 \le i \le n$, 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)

   Solve using a divide and combine algorithm. First, given a list of (x, y) coordinates, preprocess each coordinate by attaching a `cover` field, initially 0. Then, sort the coordinates by ascending x value. Break ties by ascending y value.

   Call the main function `count_cover`. Given an input of preprocessed coordinates, return the input if there are fewer than two coordinates. Else, divide the coordinates into equal left and right halves and call `count_cover` recursively on them. Name the output `left` and `right` respectively.

   Next, count coverage. Assert that no entry from `left` covers an entry from `right`. Also assert that any coverage relationships within `left` and `right` are already counted. So, only the right-side entries need to be examined. Make a pointer to the first entry in `left` and `right`. For each right-side entry, increment the left-side pointer as long as the `left` pointer is in-bounds and the `right` pointer covers the left-side entry. Then, assuming zero indexing, increase the right-side count by the index of the `left` pointer and increment the `right` entry's count.

   At this point, all coverage relationships between `left` and `right` elements are accounted for. Assert that `left` and `right` elements are each sorted by ascending y value. Using the `merge` algorithm from `mergesort`, merge `left` and `right` into a single output with entries ordered by ascending y value. Return the merged list. Pseudocode:

```
fn preprocess(coords: Vec(Coord)) -> Vec({Coord, Cover}):
    let coords_with_cover = coords.map(coord => {Coord, 0});
    sort coords_with_cover by ascending x coordinate;
    return coords_with_cover;

fn count_cover(coords: Vec({Coord, Cover})) -> Vec({Coord, Cover}):
    if coords.len < 2:
        return coords;
    let mid_ind = floor(coords.len / 2);
    let left = count_cover(coords[0..mid_ind]);
    let right = count_cover(coords[mid_ind..coords.len]);

    left left_ind = 0;
    for right_ind in 0..right.len:
        if left_ind < left.len and right[right_ind] covers left[left_ind]:
            left_ind += 1;
            continue;
        right[right_ind].cover += left_ind;

    merge left and right into ascending order by y, then x.
    return merged;
```

Correctness:
Assume inputs are preprocessed. That establishes the fact that every input has an initial coverage of 0 and that inputs are sorted by increasing x value.

The algorithm relies on multiple invariants:

1. During the divide step, every coord on the left side has an x less than or equal to every coord on the right side. Because of tie-breaking y-sorting. Any elements on the left and right sides with the same x are ordered such that the coord with a greater y is on the right.
2. In the `left` and `right` lists generated by `coord_count`, no element on the left side covers an element on the right side.
3. In the output of `count_cover`, the count associated with each coordinate accurately reflects coverage relationships with all other coordinates in the returned list.
4. The output of `coord_count` is sorted by ascending y value.

Prove these four invariants inductively by the size of n. For simplicity, assume all coordinates are distinct:

Base case 1: When $n = 1$, the input list is returned. There is no divide step, so (1) and (2) are not unsatisfied. (3) is trivially satisfied by a sorting of one element. Because a single coordinate cannot cover any coordinates, 0 remains the coverage, satisfying (4).

Base case 2: When $n = 2$, the coords are split at the middle. Because the input is sorted by x and then y, the x value of the left coord is less than or equal to the x

value of the right coord. If equal, then the left coord's y value is smaller. This satisfies (1). The recursive call just returns the coordinates. Because of (1), the x and y of the `left` coord cannot both exceed the `right` coord, so nothing in `left` can cover anything in `right`. This satisfies (2). If an element in `right` covers something in `left`, `left`'s index will stop at 1, and right's count will be incremented. This satisfies (3). Maintain (4) by using `merge` on the `left` and `right` coordinates, putting the one with a lower y first.

Inductive hypothesis: Assume the invariants hold for all lists of size $1 \leq k \leq n$.

Inductive step: Consider a list of size $2n$. Because inputs are sorted, the left and right sides are split such that the x value of every left coord is less than or equal to the x value of every right coord, with ties broken by the y value. This satisfies (1). Each sub-list is of size n. By the inductive hypothesis, the invariants hold for both `left` and `right` return results from the recursive call. Because every coord in the `left` list has either a smaller x or a smaller y (in case of tie) than every coord on the `right` list, it's definitionally impossible for a coordinate from the `left` list to cover a coordinate from the `right` list. This satisfies (2). By the inductive hypothesis, the outputs on both sides are sorted by increasing y value. Because of (2), only coords from the right side will have their counts increased at this step. Because of (1), any element from the `right` list with a greater y value than an element from the `left` list covers that element. Using two pointers in the algorithm and incrementing the left pointer to find all the left coordinates with a smaller or equal y to the current right coordinate, every right coordinate's count is correctly incremented. Again, because the entries are sorted by y value and because of (1), if a right-side entry covers up to a certain index of left-side entries, it covers every entry before the index. Through this tabulation process, (3) is established. Finally, the result values are merged. Because the two lists are already sorted by y value, only one more pass with two pointers is needed for the merge step. This establishes (4) and proves the inductive step.

Given a coordinate list of size n, the values are first preprocessed. Then, given the preprocessed list, by the inductive proof above, the returned list from `count_cover` preserves all four invariants. Because of invariant (3), every coverage relationship is accurately counted in the returned list. So, given an input list, the coverage relationships are found and correctly returned.

The only thing the algorithm returns after preprocessing is the output of `count_cover`. By the inductive proof, `count_cover` always returns the list of coordinates where (3) is preserved such that coverage is correctly counted. So, if the algorithm returns anything, it's only ever a correct counting of coord coverage. By these two cases, the algorithm must be correct.

Runtime:
Preprocessing maps n points and then sorts n points. This takes time $\Theta(n)+\Theta(n\log n)$, which yields a preprocessing time of $\Theta(n\log n)$. Consider `count_cover` as a recurrence. Given a list of size n, two recursive calls are made on lists of size $\frac{n}{2}$. Both the two-pointers tabulation and two-pointers merge always increment either the left or right index at every step and make a single pass through the inputs, so both take

time $O(n)$. This yields the recurrence $T(n) = 2T(\frac{n}{2}) + O(n)$. Apply Case II of the Master Theorem where $f(n) = O(n)$ and $\log_2 2 = 1$ such that $f(n) = \Theta(n^1 \cdot \log^0 n))$. This produces the asymptotic bound $T(n) = \Theta(n \log n)$.

Because preprocessing and the main recurrence take an asymptotically comparable amount of time, the total runtime is $\Theta(n \log n)$.

(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.

My solution is based on this subproblem structure: given a list of strings `words` and an integer `preference`, let `startline` be some index in the `words` list. The optimal pretty printing of `words` from `startline` requires finding the optimal end-of-line word `eol` after which to insert a linebreak such that the cost of the line from `startline` to `eol` plus the cost of the pretty printing beginning at `eol + 1` is minimized for all possible linebreak positions. My algorithm proceeds naturally from this idea.

The base case occurs when the last word in the paragraph is both `startline` and `eol`. Here, the cost is just that of a single word. Then, iterate `startline` backward from the last word. For each `startline`, consider each possible position of `eol` from `startline` to the end. The cost of placing a newline at the candidate position is the cost of the line from `startline` to `eol` and the cost from the word after `eol` to the end (or 0 if `eol` is the last word). Cache the most optimal `eol` position and value in `dp` at `startline` and iterate to `startline -= 1`. Once the final `startline = 0` entry is computed, the total cost of pretty printing the paragraph is stored in `dp[0].cost`, and the linebreaks can be determined by following the path of `eols` from 0 to `dp[0].eol` iteratively until the end. Here's pseudocode for such an algorithm:

```
fn cost(
    words: Vec(String),
    range_start: int,
    range_end: int,
    chars_in_range: int
) -> int:
    return result of cost function based on inputs.
           Edge case: if A >= 0 and range_end
           is the last word, the cost is 0.

fn find_pretty(words: Vec(String), pref: int) -> (total cost, linebreaks):
    let dp = Vec({ cost: inf, next_line: None } for words.len entries);

    let last_word = words.len - 1;
    let last_word_cost = cost(words, last_word, last_word, words[last_word].len);
    dp[last_word] = { cost: last_word_cost, next_line: None };

    for startline in last_word..=0:
        let chars_in_range = 0;
        for eol in startline..=max_word:
            chars_in_range += words[eol].len;
            let cost_of_interval = cost(words, startline, eol, chars_in_range);
            let cost_of_rest = dp[eol + 1].cost or 0 if eol >= last_word;
            let cost_if_break_here = cost_of_interval + cost_of_rest;
            if cost_if_break_here < dp[startline].cost:
                dp[startline] = {
                    cost: cost_if_break_here,
                    next_line: eol + 1,
                    chars_in_range
                };

    return dp[0].cost and reconstruct the line breaks by following next_line
    pointers from dp[0] until out of bounds.
```

Correctness:
Begin by proving the correctness of the recurrence by reverse induction on the position of `startline` beginning at the end:

Base case: When `startline` is the last index in the `words` list, the cost of printing that paragraph is just the cost of printing the word. This will be correctly computed by the `cost` function and is stored in the `dp` array. Note that the last-line edge case implies that this printing will have cost 0 unless the word itself is longer than the preference.

Inductive hypothesis: Assume for all `startline` positions from `words.len - 1` to some `startline = k` where `0 < k < words.len` that `dp[k].cost` is the optimal cost of pretty printing beginning at that word and that `dp[k].eol` points to the next

word break in the optimal printing.

Inductive step: Consider the pretty printing beginning at `startline = k - 1`. Such a pretty printing by-definition must include `words[startline]`, so it must be accounted for. A newline can be inserted anywhere between `startline` and the last word or not at all. For each possible placement of `eol`, the cost of printing the line from `startline` to `eol` can be computed with the `cost` function. By counting the non-whitespace characters traversed from `words[startline]` to the current `words[eol]`, the number of characters in the window is also accumulated for use in the cost function. Once the line cost from `startline` to `eol` is determined, the rest of the paragraph from `eol + 1` to the end must be considered. By the inductive hypothesis, the optimal cost of the paragraph beginning at `eol + 1` is already computed and available in the `dp` array. Because the cost of the line with `startline` is independent of the cost of the line beginning at `eol + 1`, the optimal `eol` placement is just the minimized sum of printing costs before and after the `eol`. By considering every possible place in the paragraph from `startline` to `words.len - 1` to put the `eol`, the algorithm considers every possibility and chooses the one that optimizes the total cost of pretty printing the paragraph beginning at `startline`. This proves the inductive step.

By the inductive proof above, after running the algorithm on every entry in `words` from `words.len - 1` to `startline = 0`, the entry at `dp[0]` contains the minimum cost of pretty printing the text from 0 to `words.len - 1`, which is the whole paragraph. So, given whitespace-free text where an optimal printing exists, it is returned.

Similarly, the only thing the algorithm returns is the value at `dp[0].cost` and the linebreaks stored in the `dp` array. Because these costs and `eol` selections are correct by the inductive proof above, it must be the case that only valid solutions to the input are returned from the `pretty_print` algorithm. By these two cases, the algorithm must be correct.

Runtime:
This analysis makes the assumptions that arithmetic and exponentiation (needed for the cost function) require constant time, and retrieving the number of characters in a string takes constant time. Thus, the `cost` function runs in constant time. The base case is just an array assignment, which also takes constant time. The main loop iterates the number of words in the input, which is $O(n)$. For each of those loops, the inner loop iterates $O(n)$ times again. Because the inner loop accumulates the character count of the window size as it iterates, the total cost of the main loop is thus $O(n^2)$. Retrieving the sum at the end takes constant time, and following the `eols` from `dp[0]` to the last word takes $O(n)$ time. Thus, the algorithm runs in time $O(n^2)$ where n is the number of words in the input vector.

(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.)

For $m = 40$, I computed an optimal cost of 396 with lines beginning at indices 0, 5, 10, 18, 24, 30, 40, 48, 54, 62, 71, 78, 85, 92, 99, 105, 112, 120, 127, 135, 142.

```
########################################
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.)
```

For $m = 72$, I computed an optimal cost of 99 with lines beginning at indices 0, 10, 22, 37, 50, 64, 79, 92, 104, 117, 132.

```
########################################################################
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 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 $p_1, \ldots, p_n, q_0, \ldots, q_n$.

> First, I'll make some assumptions and definitions: For each adjacent available books x_1 and x_2, the corresponding range of unavailable books can be identified and indexed in constant time by the exclusive range $(x1, x2)$. For brevity, name the set of available books `av` and the set of unavailable ranges `unav`. Also, define a variable name `ev_pathlen` to represent the expected value number of nodes traversed to service a given query. My pseuducode also uses a `dp` hash map with keys indexed by a minimum value, maximum value, and tree depth. Tree depth is synonymous with pathlen.
>
> Consider this algorithm: Given a `range` and `depth`, construct a key from the range and depth and return the value in `dp` if one already exists. Else, if the range is in `unav`, then the range defines a leaf node. Create a leaf node from the range, compute the `ev_pathlen` as the probability of the leaf node times the depth, and store that in `dp`. Return that result. Otherwise, this isn't a base case, and an optimal result from the range must be computed. For each book in `av`, consider the case where that entry is the root of a tree. Because it splits the range, recursively compute the left side from the current range min to the root's min and the right side from the root's max to the current range max. Both use the current depth plus 1 as the `depth` parameter. Assuming the optimality of the function, the `left` and `right` return values represent the most optimal trees in the given range, and their `ev_pathlen` values reflect the expected value of having that subtree at `depth + 1`. Choose the pairing of `root`, `left`, and `right` that minimizes total `ev_pathlen`, store that result in `dp`, and return it. By calling that function initially with a range from $-$inf to inf, a depth of `1`, and an empty `dp` hashmap, the returned value will be the root of the optimal tree and the expected number of nodes traversed to service a given query. Pseudocode:

```
fn optimal_tree(
    range: (int, int),
    depth: int,
    dp: HashMap((min, max, depth), (TreeNode, ev pathlen))
) -> (TreeNode, prob, height):
    let key = (range.min, range.max, depth);
    if key in dp:
        return dp[key];

    if range in unav:
        let leaf = TreeNode(unav[range], None, None);
        dp[key] = (leaf, unav[range].prob * depth);
        return dp[key];

    let best_root = None;
    let best_ev_pathlen = inf;
    for new_root in av.filter(within range):
        let left = optimal_tree((range.min, new_root.min), depth + 1, dp);
        let right = optimal_tree((new_root.max, range.max), depth + 1, dp);
        let root_ev_pathlen = (new_root.prob * depth) + left.ev_pathlen + right.ev_pathlen;
        if root_ev_pathlen < best_ev_pathlen:
            best_root = TreeNode(new_root, left.root, right.root);
            best_ev_pathlen = root_ev_pathlen;

    dp[key] = (best_root, best_ev_pathlen);
    return dp[key];
```

Correctness:
The algorithm makes multiple claims, proven by induction:

- The algorithm generates a binary search tree with exactly all entries in both av and unav.
- The tree generated by `optimal_tree` optimally minimizes the expected number of nodes that must be traversed to service a query.
- The value at `(min, max, depth)` in `dp` contains the root of the tree that minimizes the expected number of nodes traversed to service a query on a tree with the given range and depth.

Base case: When the range is in `unav`, the range indicates a leaf node by definition of the question. For a leaf node at a given depth, the expected value of nodes traversed to reach that leaf is just the probability of the leaf times the number of nodes traversed. By storing the leaf node and `ev_pathlen` in `dp` and returning the value in `dp`, the algorithm correctly handles the base case.

Inductive hypothesis: Assume the claims hold for recursive calls to sub-trees splitting the current range with a depth d of $d+1$.

Inductive step: At depth d, if the key is in `dp`, assert by the inductive hypothesis that the cached value is the correct return value. Otherwise, the `for` loop is entered. Because

the range is not terminal, there are entries in `av` to iterate within the range. For each potential root in the range, the left and right subtrees are retrieved. By the inductive hypothesis, the return values of these recursive calls are optimal. If root is on the far-left of the range, all nodes in the sub-range/subtree for the range will be on the right side. Vice versa. This loop considers all possible trees for this range and depth. By only permitting ranges smaller than `range.min` on the left and ranges greater than `range.max` on the right, the BST property is maintained. By permitting every node in the range to be considered and only narrowing the sub-range by the single value chosen as the root, every sub-node is recursively added to the tree in optimal order. Finally, consider the `ev_pathlen` values returned from each recursive call. Each `ev_pathlen` is a summation of probability times depth. By linearity of expectation, the summation of probabilites on the left and right plus root is the probability of any node in the tree beginning at root being visited. Because each node's probability is weighted by its depth in the tree, the `ev_pathlen` computed by combining those of left and right with the expected value of ending a search at the root is the total expected value path length for queries served by the tree at depth d. By minimizing this, the algorithm chooses the tree that minimizes expectation of path length for the current root and all subtrees. By these observations, the inductive step must be correct.

By calling the recursive function on a range from $-$inf to inf with an initial depth of 1, every node in every range is considered, and, by the inductive proof above, a tree is returned that minimizes the expected number of nodes traversed to serve a query. Because of this, given inputs as-defined above, the output is a tree that optimizes path length.

Similarly, the only thing the algorithm returns is the value in `dp` for a given key. By The inductive proof above, the value at `dp` is the optimal tree for the given range and depth. From the initial range and depth of $-$inf, inf, 1, the only value returned from the algorithm is a tree that minimizes the expected number of nodes traversed to serve a query on the entire range of books.

Runtime:
For a given range and depth, if the range and depth has already been computed, it exists in `dp` and is returned in constant time. The only non-constant work in this algorithm is performed when an entry is not yet in `dp`. In such a case, the algorithm performs $O(n)$ work to call subproblems. Because these subproblems recursively explore and fill the entire state space of `dp`, the time complexity of the algorithm is limited by the maximum size of `dp`. By the question definition, the range $-$inf to inf can be split into n sub-ranges by the values in `av`. Considering all (valid) ranges in the problem requires $O(n^2)$ time. The tree depth is at-most the number of nodes in the tree, which is $O(n)$. So, in total, the algorithm runs in $O(n^3)$ time.