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 pre-processing 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)

   (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 $A \ge 0$ and $2^{-A} - A^3 - 1$ if $A \le 0$. 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 $A \ge 0$. 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.

   (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.)

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