CS 577: Introduction to Algorithms Fall 2022

Homework 1

Instructor: Dieter van Melkebeek TA: Nicollas Mocelin Sdroievski

This homework covers the divide and conquer paradigm. **Problem 3 must be submitted for grading by 2:29pm on 9/20.** Please refer to the homework guidelines on Canvas for detailed instructions.

Warm-up problems

1. Consider the problem of powering an integer:

   Input: (a, b) with $a, b \in \mathbb{Z}$ and $b \geq 1$

   Output: a^b, which we define as $a^b \doteq \underbrace{a \cdot a \cdot \cdots \cdot a}_{b \text{ times}}$, where "$\cdot$" denotes multiplication.

   Design an algorithm for this problem that uses at most $O(\log b)$ multiplications.

2. A Toeplitz matrix is a matrix A in which the value of the (i, j)th entry A_{ij} only depends on the value of $i - j$.

   Design an algorithm that computes the product Ax of a Toeplitz matrix $A \in \mathbb{R}^{n \times n}$ with a vector $x \in \mathbb{R}^n$ using $O(n \log n)$ elementary operations.

Regular problems

3. [Graded] You are given a perfect binary tree T with $n = 2^d$ leaves, where each leaf contains an integer value. Reading the leaf values from left to right yields a sequence of integers. The question is how small we can make the number of inversions in that sequence by applying any number of operations of the following type: Select an internal vertex and swap the two child subtrees. Data associated to a vertex in a subtree follow the vertex in the swap.

   For example, if the sequence of leaf values is $(4, 2, 1, 3)$, then a swap at the root followed by a swap at the right child of the root turns the sequence into $(1, 3, 2, 4)$, which has only one inversion. See the figure below. It is impossible to do better, so the answer for this particular example is 1.

   

   Design an $O(n \log n)$ algorithm for this problem.

4. Consider the following computational problem:

   Input: Array $A[1, \ldots, n]$ of positive integers.

   Output: Array $C[1, \ldots, n]$ where $C[i]$ is the number of $j \in \{1, \ldots, i-1\}$ with $A[j] \geq A[i]$.

   For example, if $A = [8, 12, 10, 9, 10, 12, 7]$ then $C = [0, 0, 1, 2, 2, 1, 6]$.

   Design an $O(n \log n)$ algorithm for this problem.

5. You are given a string $T[1 \ldots n]$ over a finite alphabet A, where A does not contain the symbol $*$, and a string $P[1 \ldots m]$ of length $m \leq n$ over the alphabet $A \cup \{*\}$. Your goal is to find all the occurrences of P in T, where the symbol $*$ acts as a wildcard, i.e., it matches every symbol in A.

   For example, for $T = (a, a, b, a, b, a, a)$ and $P = (*, a, b)$, there are two occurrences, namely $T[1 \ldots 3] = (a, a, b)$ and $T[3 \ldots 5] = (b, a, b)$.

   Design an algorithm that outputs the start positions of all occurrences of P in T and uses $O(n \log n)$ elementary operations.

   In the above example, the output would be $(1, 3)$.

Challenge problem

The following is one of the nicest introductory algorithm problems I know. Give it a try!

6. You are given n coins, at least one of which is bad. All the good coins weigh the same, and all the bad coins weigh the same. The bad coins are lighter than the good coins.

   Design an algorithm that makes $O((\log n)^2)$ weighings on a balance to find the exact number of bad coins. Each weighing tells you whether the total weight of the coins on the left side of the balance is smaller than, equal to, or larger than the total weight of the coins on the right side.

Programming problem

7. SPOJ problem Insertion Sort (problem code CODESPTB).

CS 577: Introduction to Algorithms Fall 2022

Homework 1 Solutions to Warm-up Problems

Instructor: Dieter van Melkebeek TA: Nicollas Mocelin Sdroievski

Problem 1

> Consider the problem of powering an integer:
> **Input:** (a, b) with $a, b \in \mathbb{Z}$ and $b \geq 1$
> **Output:** a^b, which we define as $a^b \doteq \underbrace{a \cdot a \cdot \cdots \cdot a}_{b \text{ times}}$, where "$\cdot$" denotes multiplication.
> Design an algorithm for this problem that uses at most $O(\log b)$ multiplications.

We first consider the case where b is a power of two. The key idea is to treat the expression

$$\underbrace{a \cdot a \cdot \cdots \cdot a}_{b \text{ times}}$$

like an array to be divided into halves. We can rewrite it as

$$(\underbrace{a \cdot a \cdot \cdots \cdot a}_{b/2 \text{ times}}) \cdot (\underbrace{a \cdot a \cdot \cdots \cdot a}_{b/2 \text{ times}}),$$

and observe that the factors are equal. Because they are equal, we can compute both of them with only one recursive call. Since $b/2$ remains a power of two, this is a complete algorithm for that case.

For the general case, b is an arbitrary positive integer. Still we can use a similar decomposition into halves:

$$a^b = (\underbrace{a \cdot a \cdot \cdots \cdot a}_{\lfloor b/2 \rfloor \text{ times}}) \cdot (\underbrace{a \cdot a \cdot \cdots \cdot a}_{\lceil b/2 \rceil \text{ times}}).$$

In this case, the second part is identical to the first part up to a single factor of a. The extra factor is present if and only if b is odd. So we can recursively compute the first part, square it, and then, when b is odd, multiply in one more factor of a. This gives us Algorithm 1:

It remains to prove that Algorithm 1 is correct, and to analyze its running time. Correctness is formally argued by induction, and we provide some of the details. It is easy to see that the algorithm is correct for $b = 1$, since $a^b = a$. For $b > 1$, we consider two cases:

1. b is even. In this case, $a^b = a^{\lfloor b/2 \rfloor} \cdot a^{\lfloor b/2 \rfloor}$.
2. b is odd. In this case, $a^b = a^{\lfloor b/2 \rfloor} \cdot a^{\lfloor b/2 \rfloor} \cdot a$.

In both cases, substituting $c = a^{\lfloor b/2 \rfloor}$ shows that the algorithm returns the correct value.

As for the running time, we use the recursion tree method. Each invocation of Fast-Power has at most one recursive call, so the shape of the recursion tree is a line. When moving one level down the tree, the 'b' argument is halved (rounding down). It becomes 1 after $\lfloor \log b \rfloor$ halvings(essentially by the definition of logarithm). As a result, the depth of the recursion tree is at most $\lfloor \log b \rfloor$. Each node in the recursion tree does at most two (hence $O(1)$-many) multiplications. It follows that there are at most $O(1) \cdot \lfloor \log b \rfloor = O(\log b)$ multiplications in any invocation of Fast-Power.

Algorithm 1
Input: $a, b \in \mathbb{Z}, b \geq 1$
Output: a^b

```
1: procedure FAST-POWER(a, b)
2:     if b = 1 then
3:         return a
4:     else
5:         c ← FAST-POWER(a, ⌊b/2⌋)
6:         if b is even then
7:             return c · c
8:         else
9:             return c · c · a
```

Problem 2

A Toeplitz matrix is a matrix A in which the value of the (i,j)th entry A_{ij} only depends on the value of $i-j$.

Design an algorithm that computes the product Ax of a Toeplitz matrix $A \in \mathbb{R}^{n\times n}$ with a vector $x \in \mathbb{R}^n$ using $O(n \log n)$ elementary operations.

Let's consider what the product of a Toeplitz matrix and a vector looks like. An $n \times n$ Toeplitz matrix has $2n-1$ distinct values, one for each diagonal of the matrix. Call our Toeplitz matrix T. The value for entry (i,j) is t_{i-j}. That is, T looks like this:

$$\begin{bmatrix} t_0 & t_{-1} & \cdots & t_{-n+2} & t_{-n+1} \\ t_1 & t_0 & \cdots & t_{-n+3} & t_{-n+2} \\ \vdots & \ddots & \ddots & \ddots & \\ t_{n-2} & t_{n-3} & \cdots & t_0 & t_{-1} \\ t_{n-1} & t_{n-2} & \cdots & t_1 & t_0 \end{bmatrix}$$

We want to multiply T by an n-vector x, with values $x_0, x_1, \cdots, x_{n-1}$. The product is then given by the n-dimensional vector

$$\begin{bmatrix} t_0 \cdot x_0 + t_{-1} \cdot x_1 + \cdots + t_{-n+1} \cdot x_{n-1} \\ t_1 \cdot x_0 + t_0 \cdot x_1 + \cdots + t_{-n+2} \cdot x_{n-1} \\ \vdots \\ t_{n-1} \cdot x_0 + t_{n-2} \cdot x_1 + \cdots + t_0 \cdot x_{n-1} \end{bmatrix}$$

To solve this problem, we construct two polynomials – one from T and one from x – such that by looking at the product of these polynomials we can easily find the entries in the vector Tx.

We construct the first polynomial, of degree $2n-1$, from the values of T:

$$T(z) = t_{-n+1} + t_{-n+2}z + t_{-n+3}z^2 + \cdots + t_0 z^{n-1} + t_1 z^n + \cdots + t_{n-2}z^{2n-3} + t_{n-1}z^{2n-2}$$

The other polynomial, of degree $n-1$, we get from the values of x:

$$x(z) = x_0 + x_1 z + x_2 z^2 + \cdots + x_{n-1}z^{n-1}$$

We claim that every entry of the vector Tx is the coefficient of a term in the product of these two polynomials. In particular, the 0-th entry of Tx is the coefficient of the term of degree $n-1$ in $T(z) \cdot x(z)$, the "first" entry of Tx is the coefficient of the term of degree n, and, in general, the i-th entry of Tx (for $0 \le i \le n-1$) is the coefficient of the term in $T(z) \cdot x(z)$ of degree $i+n-1$. Indeed, the coefficient for this term is $\sum_{j=0}^{n-1} x_j t_{i-j}$, which is precisely the i-th entry of the product vector.

We now use the fact that multiplication of polynomials of degree at most d can be done in time $O(d \log d)$ in the standard coefficient representation, which allows us to compute the product of $T(z)$ and $x(z)$ in time $O(n \log n)$ since they have degree $2n-2$ and $n-1$, respectively. All that is left then is to read off the coefficients from the result, which takes linear time, so the total running time is $O(n \log n)$.

CS 577: Introduction to Algorithms Fall 2022

Homework 1 Solutions to Regular Problems

Instructor: Dieter van Melkebeek TA: Nicollas Mocelin Sdroievski

Problem 3

You are given a perfect binary tree T with $n = 2^d$ leaves, where each leaf contains an integer value. Reading the leaf values from left to right yields a sequence of integers. The question is how small we can make the number of inversions in that sequence by applying any number of operations of the following type: Select an internal vertex and swap the two child subtrees. Data associated to a vertex in a subtree follow the vertex in the swap.

Design an $O(n \log n)$ algorithm for this problem.

We can equivalently think of the input as the array A of leaf values, and the objective as applying some among a restricted family of permutations to A so as to minimize the number of inversions in A. The inversions in A can be broken into three types:

(i) inversions within the left half of A,

(ii) inversions within the right half of A, and

(iii) inversions that cross the boundary between the two halves.

Swap operations on nodes of the left subtree of T only affect (i). Similarly, swap operations on the right subtree of T only affect (ii). A swap on the root of T only affects (iii). This means that in order to minimize the total count of inversions in T, we can independently minimize the counts of (i), (ii), and (iii).

Minimizing the counts of (i) and (ii) corresponds to simpler instances of the given problem, namely for the left subtree of T and the right subtree of T, respectively; thus we can find those quantities by recursion.

Minimizing (iii) is where the real work happens. Given the recursion into (i) and (ii), we can afford $O(n)$ work to compute (iii) and still get an $O(n \log n)$ running time.

Recall the procedure Count-Cross from class. It takes two sorted arrays as input, and, in linear time, outputs the number of inversions in the concatenation of the first and the second array. Thus, if L and R are sorted copies of the left and right halves of A, then we can compute the minimum value of (c) as the minimum of Count-Cross(L,R) and Count-Cross(R,L). So it suffices to get our hands on L and R.

To do that, we strengthen the specification of our algorithm. In addition to computing the minimum number of inversions, we require that it return a sorted copy of A. Thus, when we recurse on the left half of A and the right half of A, these recursive calls directly return L and R as well. As discussed, this allows us to minimize the counts of (i), (ii), and (iii), but now we must also compute a sorted copy of A. We can meet this requirement by using the procedure Merge from class. It takes as input two sorted arrays and returns their sorted concatenation. Since we have L and R, we can use Merge to compute a sorted copy of A. Merge runs in linear time.

All together, we get a divide-and-conquer algorithm for the augmented problem, where in addition to the minimum number of inversions for T we also output the leaf values in sorted order.

The algorithm is given as Algorithm 1 below. The answer to the original question is the first component of the value returned by TreeInversionCount(T), where T is the original input tree.

Algorithm 1 Counting Tree Inversions

Input: T, a perfect binary tree with an integer at each leaf
Output: The pair (v, S), where v is the minimum possible number of inversions in T's leaf node values after swapping subtrees of some internal nodes, and S is a sorted array of all leaf node values in T

```
1:  procedure TreeInversionCount(T)
2:      if |T| = 1 then
3:          return (0, k) where k is the integer stored at the node in T
4:      (a, L) ← TreeInversionCount(T's left subtree)
5:      (b, R) ← TreeInversionCount(T's right subtree)
6:      c1 ← Count-Cross(L, R)
7:      c2 ← Count-Cross(R, L)
8:      c ← min(c1, c2)
9:      v ← a + b + c
10:     S ← Merge(L, R)
11:     return (v, S)
```

Correctness. Correctness follows from an inductive argument on the depth of T. The base case is when $d = 0$, i.e, T has only one node. In this case, TreeInversionCount returns 0 and a singleton array of that element. Since a sequence of one value never has any inversions, and an array of one element is always sorted, this is the correct result.

We now establish the inductive step. Recall that in order to minimize the total number of inversions, it suffices to compute the minimum counts of inversions of types (i), (ii), and (iii) independently. By the inductive hypothesis, TreeInversionCount returns correctly for T's left and right children. Thus a stores the minimum count of inversions of type (i), and b stores the minimum count of inversions of type (ii). L and R store sorted copies of the left and right halves of A, respectively. As discussed, the number of inversions of type (iii) equals the number of inversions in LR or in RL, depending on whether a swap is done at the root. By correctness of Count-Cross, c_1 and c_2 store these respective values, and so $c = \min(c_1, c_2)$ is the minimum count of inversions of type (iii). It follows that $a+b+c$ is the minimum possible number of all kinds of inversions, and so v is a valid output for the first part of the spec. By correctness of Merge, S correctly stores a sorted copy of A, so is a valid output for the second part of the spec. Thus TreeInversionCount correctly implements its specification.

Analysis. The running time analysis is nearly identical to the analysis for Merge Sort. In the recursive case, any given execution of TreeInversionCount makes two recursive calls, two calls to Count-Cross, a call to Merge, and $O(1)$ additional work. The recursive calls of TreeInversionCount divide the input into equal-size halves. As discussed in class, Count-Cross and Merge take time linear in the sum of the lengths of their arguments; it follows that each level of the recursion tree does a combined $O(n)$ work. As there are $O(\log n)$ levels, the running time of TreeInversionCount is $O(n \log n)$.

Problem 4

Consider the following computational problem:

Input: Array $A[1, \ldots, n]$ of positive integers.

Output: Array $C[1, \ldots, n]$ where $C[i]$ is the number of $j \in \{1, \ldots, i-1\}$ with $A[j] \ge A[i]$.

Design an $O(n \log n)$ algorithm for this problem.

We use a divide-and-conquer approach, in which we break up the given array $A[1, \ldots, n]$ into two halves, $L \doteq A[1, \ldots, \lfloor n/2 \rfloor]$ and $R \doteq A[\lfloor n/2 \rfloor + 1, \ldots, n]$, recursively find the respective solutions C_L and C_R for those subproblems, and then use C_L and C_R to efficiently compute the solution C for A. Below, we show how to compute C in $O(n)$ time, given C_L and C_R. Given that, the resulting algorithm runs in time $O(n \log n)$ as it follows the same pattern as MergeSort.

The first half of C equals C_L, i.e., $C[1, \ldots, \lfloor n/2 \rfloor] = C_L[1, \ldots, \lfloor n/2 \rfloor]$. This is because the elements that precede elements in the first half of C all occur in that first half, which equals C_L.

For the second half of C, we need to add to C_R the number of elements in L that are larger than or equal to the element from A under consideration. More precisely, for $1 \le k \le \lceil n/2 \rceil$, we have:

$$C[\lfloor n/2 \rfloor + k] = C_R[k] + |\{i \in [\lfloor n/2 \rfloor] \text{ such that } L[i] \le R[k]\}| \,.$$

Note that the added term equals the number of inversions formed by the k-th element of R with elements from L in the concatenation LR, provided that we also count pairs of positions that contain equal values as inversions. In class we saw a linear-time algorithm to count the aggregate of all such inversions (and sort the concatenation LR) when both L and R are sorted in nondecreasing order. We use the same procedure here with the following modifications:

- Consider cross pairs of equal values as inversions. This can be achieved by moving the pointer into R rather than the pointer into L in the case of equal values.
- Record separately the counts for each element of R rather than aggregating them.
- Keep track of the original positions of the sorted entries in the given array A. This enables us to add the cross inversions to the correct positions.

For clarity we include pseudocode for the combining process IndividualMerge. It uses records consisting of the value of an entry of the given array and the position of the entry in the given array in order to represent the given array.

For completeness we also include pseudocode for the recursive procedure CountIndividualInversions that takes an array and returns the array in sorted record format as well as the solution to the problem. See pages 4–5.

Correctness Correctness of CountIndividualInversions and IndividualMerge is established by organizing the ideas from the above discussion into a typical correctness proof.

To start, we establish correctness of CountIndividualInversions by induction on n. The n-th statement to be established is that CountIndividualInversions matches its specification on all inputs A of length n. As base cases, we have $n = 0$ and $n = 1$. In the former case, we are required to return two empty arrays, as the algorithm does. In the latter case, we are required to return an array containing the pair $(1, A[1])$, and an array containing 0, as the algorithm does.

For $n > 1$, CountIndividualInversions enters its recursive case. The specification for the recursive calls and the specification for IndividualMerge compose to match the specification for CountIndividualInversions, so correctness follows directly from the inductive hypothesis and correctness of IndividualMerge.

Finally, it remains to establish correctness of IndividualMerge. However, due to its similarity to correctness of Count-Cross from lecture/scribe notes, we omit the details here.

Running Time Lastly, we need to analyze the running time of CountIndividualInversions (hence also IndividualMerge). As with Count-Cross, IndividualMerge runs in time $O(\ell+r)$. CountIndividualInversions has a recursion tree shaped identically to MergeSort, as well as the same work-per-node up to constant factors, so it runs in time $O(n \log n)$.

Algorithm 2

Input: Array $A[1, \ldots, n]$ of intergers.
Output: $(C, \widehat{A})$ where C is the solution for A, and $\widehat{A}$ contains the pairs $(i, A[i])$ for $i = 1, \ldots, n$, sorted in nondecreasing order of the second component.

```
1: procedure CountIndividualInversions(A[1,...,n])
2:     if n = 0 then
3:         return ( [], [] )
4:     else if n = 1 then
5:         return ( [(1,A[1])], [0] )
6:     else
7:         (C_L, Â_L) ← CountIndividualInversions(A[1,...,⌊n/2⌋])
8:         (C_R, Â_R) ← CountIndividualInversions(A[⌊n/2⌋+1,...,n])
9:         return IndividualMerge(Â_L, Â_R, C_L, C_R)
```


Algorithm 3

Input: Arrays $\widehat{L}[1, \ldots, \ell]$ and $\widehat{R}[1, \ldots, r]$ of (position, value) records corresponding to arrays L and R, respectively; L and $\widehat{R}$ are sorted in order of nondecreasing value. Solutions $C_L[1, \ldots, \ell]$ and $C_R[1, \ldots, r]$ for the arrays L and R, respectively.
Output: $(C, \widehat{A})$ where $\widehat{A}$ is an array of (position, value) records corresponding to the concatenation LR of L and R, sorted in nondecreasing order of value, and C is the solution for LR.

```
1: procedure IndividualMerge(L̂[1,ℓ], R̂[1,r], C_L, C_R)
2:     i ← 1; j ← 1; k ← 1
3:     C ← C_L C_R (concatenation)
4:     while j ≤ r do
5:         if i ≤ ℓ and L̂[i].value < R̂[j].value then
6:             Â[k] ← L̂[i]
7:             i ← i + 1
8:         else
9:             Â[k] ← R̂[j] with 'position' component increased by ℓ
10:            C[Â[k].position] ← C[Â[k].position] + (ℓ − i + 1)
11:            j ← j + 1
12:        k ← k + 1
13:    if i ≤ ℓ then
14:        Â[k,...,ℓ+r] ← L̂[i,...,ℓ]
15:    return (Â[1,...,ℓ+r], C[1,...,ℓ+r])
```


Problem 5

You are given a string $T[1 \ldots n]$ over a finite alphabet A, where A does not contain the symbol *, and a string $P[1 \ldots m]$ of length $m \leq n$ over the alphabet $A \cup \{*\}$. Your goal is to find all the occurrences of P in T, where the symbol * acts as a wildcard, i.e., it matches every symbol in A.

Design an algorithm that outputs the start positions of all occurrences of P in T and uses $O(n \log n)$ elementary operations.

A natural (but inefficient) way of solving this problem is as follows: for every starting position $1 \leq i \leq n - m$, we compare $T[i]$ with $P[1]$, $T[i+1]$ with $P[2]$ and so on, keeping track of positions i for which we obtain perfect matches (recall that $*$ matches every symbol in A). This approach takes time $O(nm) = O(n^2)$, which is too much, but it gives us some insight into the more efficient solution for this problem. Say T and P are vectors of coefficients for two polynomials $T(x)$ and $P(x)$, then, polynomial multiplication does something similar: it fixes a vector, say T and computes inner products for all possible shifts of (the reverse of) the other vector, in this case P. To understand how this can be useful, consider the following example:

$$T = (a, a, b, b, a) \text{ and } P = (a, b, b),$$

which we view as the polynomials

$$T(x) = a + ax + bx^2 + bx^3 + ax^4 \text{ and } P^R(x) = b + bx + ax^2.$$

Notice that we reversed P so that the inner products align correctly (instead of in reverse). By assigning numeric (possibly complex) values to each symbol in A, $T(x)$ and $P^R(x)$ are polynomials of degree $n-1$ and $m-1$, respectively. Their product is

$$C(x) = ab + (ab + ab)x + (a^2 + ab + b^2)x^2 + (a^2 + b^2 + b^2)x^3 + (ba + b^2 + ab)x^4 + (ab + ba)x^5 + a^2x^6.$$

Note that the coefficient of x^3 being $a^2 + b^2 + b^2$ indicates that we matched P with the substring of T of size 3 that ends in position 4 (since the coefficient of x^i is given by the $(i+1)$-th position in $C(x)$, when viewed as a vector). This is the only match for these two strings, and the only time where the expression $a^2 + b^2 + b^2$ appears as a coefficient in $C(x)$.

Now, our objective is to assign numeric (possibly complex) values to each of $a, b, c, \cdots \in A$ such that we can determine when a sum is composed of matching square symbols, e.g. $a^2 + b^2 + \ldots$. We still need to deal with the wildcard symbol $*$, but that will not be too complicated. One way we can try to differentiate between sums of squares and mismatched sums is by forcing $a^2 = b^2 = c^2 = \cdots = 1$ for all symbols in A, while having the real part of $ab, ac, bc, \ldots$ be less than 1, again for all pairs of symbols in A. If we can do this, then a matching sum will add up to exactly the length of P (assuming no wildcards), and sums with mismatched symbols will add up to a number whose real part is less than that. To deal with wildcards, we set the value of $*$ to 0, and only require that the sum equals the number of non-wildcard symbols in P.

Let $|A| = k$. To achieve our objective, we map each symbol in T to a distinct k-th root of unity. To be more precise, for $A = \{a_0, a_1, \ldots, a_{k-1}\}$, we map each occurrence of a_j in T to the complex number ω_k^j, where ω_k denotes a primitive k-th root of unity. Similarly, we map each occurrence of

a_j in P to the multiplicative inverse of ω_k^j, that is, we map a_j to ω_k^{k-j} (note that $\omega_k^j \cdot \omega_k^{k-j} = \omega_k^k = 1$). That way, when we multiply two matching symbols (one in T and one in P), the result is exactly 1, but when we multiply mismatched symbols we get a result whose real part is less than 1 (as the result lies on the unit circle but is different from 1).

The final algorithm is then as follows, on input $T[1, \ldots, n]$ over an alphabet A of size k, and $P[1, \ldots, m]$ over $A \cup \{*\}$ where $m \leq n$, do the following:

1. Construct the polynomial $T(x)$ of degree $n-1$ with coefficients given by mapping each occurrence of a_j in T (where $a_j \in A$) to ω_k^j
2. Construct the polynomial $P^R(x)$ of degree $m-1$ with coefficients given by mapping each occurrence of a_j in P^R (the reverse of P) to ω_k^{k-j} and mapping $*$ to 0.
3. Compute the product $C(x) = T(x) \cdot P^R(x)$.
4. Let L be an empty list.
5. Let p be the number of non-wildcard symbols in P. For each $j \in \{m-1, \ldots, n-1\}$ such that the coefficient of x^j in $C(x)$ equals p, add $j - m + 2$ to L.
6. Return L.

The reason we return a list with positions $j - m + 2$ instead of j is that the problem statement asks for the start positions of all occurrences of P in T, while our approach gives us the ending positions minus one (because of the conversion from arrays to polynomials).

Correctness for this procedure follows from the preceding discussion. The running time, in terms of elementary operations, is dominated by computing the product $C(x) = T(x) \cdot P^R(x)$, which takes time $O(n \log n)$ using the polynomial multiplication algorithm from class since the degree of $T(x)$ and $P^R(x)$ is at most n.