CS/Math 473 ✧ Fall 2022 **Midterm 1 Solutions Problem 1**	Name:

Suppose we are given a sorted array $A[1..n]$ of n distinct integers, which are not necessarily positive.

(a) Describe a fast algorithm that either computes an index i such that $A[i] = i$ or correctly reports that no such index exists.

(b) Suppose we know in advance that $A[1] > 0$. Describe an even faster algorithm that either computes an index i such that $A[i] = i$ or correctly reports that no such index exists.

Solution (part (a)): We can solve the problem in ***$O(\log n)$ time*** using a variant of (or a reduction to) binary search. Here are two pseudocode descriptions of the algorithm, one recursive and one iterative.

```
⟨⟨Find index j such that i ≤ j ≤ k and A[j] = j⟩⟩
FindIndex(i, k):
  if i > k
      return None
  j ← ⌈(i + k)/2⌉
  if A[j] = j
      return j
  else if A[j] > j
      return FindIndex(i, j − 1)
  else
      return FindIndex(j + 1, k)
```

```
FindIndex(A[1..n]):
  lo ← 1; hi ← n
  while lo ≤ hi
      mid ← ⌈(lo + hi)/2⌉
      if A[mid] = mid
          return mid
      else if A[mid] > mid
          hi ← mid − 1
      else
          lo ← mid + 1
  return None
```

The key observation is that because A is a *sorted* array of *distinct* integers, we have $A[j] \geq A[i] + (j - i)$ for all indices $i < j$. In particular, if $A[i] > i$, then $A[j] > j$ for all $j > i$.

Equivalently, suppose we (implicitly) define a new array $B[1..n]$ by setting $B[i] = A[i] - i$ for all i. Then the elements of B are sorted in *non-decreasing* order (but they are not necessarily distinct), and we are looking for an index i such that $B[i] = 0$. ■

Rubric: 8 points max. For an explicit algorithm: 1 for binary search idea + 1 for base case + 4 for recursive cases + 2 for time analysis. −1 for each off-by-one error. −1 for returning True/False instead of index. −1 for stating running time as a recurrence without solving it. A reduction to binary search is worth full credit. Max 3 points for a $\Theta(n)$-time algorithm; max 2 points for anything slower; scale partial credit.

Solution (part (b)): If $A[1] = 1$, we can clearly return 1 immediately. On the other hand, if $A[1] > 1$ then $A[i] > i$ for all i, so we can return None immediately. So we can solve this problem in ***$O(1)$ time***! ■

Rubric: 2 points: 1 for algorithm + 1 for running time

CS/Math 473 ✧ Fall 2022 Midterm 1 Solutions Problem 2	Name:

Describe and analyze an algorithm to find a matching with maximum total weight, in a binary tree with weighted edges.

Solution: Let T be the input tree, and let r denote its root node. For any vertex v, we define two functions:

- $MWM(v)$ is the weight of a maximum weight matching in the subtree rooted at v.
- $MWM^*(v)$ is the weight of a maximum weight matching in the subtree rooted at v, *where v is not incident to a matching edge*.

We need to compute $MWM(r)$. These functions satisfy the following mutual recurrences. Here $v.l$ and $v.r$ denote the left and right children of v, respectively.

$$MWM(v) = \begin{cases} 0 & \text{if } v = \textsc{Null} \\ 0 & \text{if } v \text{ is a leaf} \\ \max\left\{ \begin{array}{c} w(v, v.l) + MWM^*(v.l) + MWM(v.r) \text{ if } v.l \neq \textsc{Null} \\ w(v, v.r) + MWM(v.l) + MWM^*(v.r) \text{ if } v.r \neq \textsc{Null} \\ MWM^*(v) \end{array} \right\} & \text{otherwise} \end{cases}$$

$$MWM^*(v) = \begin{cases} 0 & \text{if } v \text{ is a leaf} \\ MWM(v.l) + MWM(v.r) & \text{otherwise} \end{cases}$$

We can memoize these functions into two new fields at each node of T, and we can evaluate the functions in postorder.

The resulting dynamic programming algorithm runs in ***$O(n)$ time***, where n is the number of vertices in T. ■

Rubric: 10 points: standard dynamic programming rubric. These are not the only correct solution.
−1 for assuming every interior node has both left and right children.

CS/Math 473 ✧ Fall 2022 **Midterm 1 Solutions Problem 3**	Name:

Suppose we are given two bit strings $P[1..m]$ (the "pattern") and $T[1..n]$ (the "text"), where $m \le n$. Describe and analyze an algorithm to find the minimum Hamming distance between P and a substring of T of length m. For full credit, your algorithm should run in $O(n \log n)$ time.

Solution (consider 0s and 1s separately): For any integer $0 \le s \le n-m$ ("shift"), let $HD(s)$ denote the Hamming distance between $P[1..m]$ and $T[s+1..s+m]$; we need to compute $\min_s HD(s)$.

Let us write $HD(s) = m - Both1(s) - Both0(s)$, where

- $Both1(s)$ is the number of indices i such that $P[i] = T[i+s] = 1$.
- $Both0(s)$ is the number of indices i such that $P[i] = T[i+s] = 0$.

More formally, we have

$$Both1(s) = \sum_{i=1}^{m} P[i] \cdot T[s+i] \qquad Both0(s) = \sum_{i=1}^{m} (1 - P[i]) \cdot (1 - T[s+i])$$

We can evaluate $Both1(s)$ for every index s using convolution as follows. Define two sequences p and t by setting $p_i = P[m-i]$ and $t_i = T[i]$ for each index i. Then for all s we have

$$Both1(s) = \sum_i p_{m-i} \cdot t_{s+i} = (p \star t)_{m+s}$$

We can construct the sequences p and t in $O(n)$ time, and then compute their convolution in $O(n \log n)$ time using fast Fourier transforms.

We can similarly express $Both0(s)$ as a convolution by defining $p'_i = 1 - P[m-i]$ and $t'_i = 1 - T[i]$ for each index i. Then for all s we have

$$Both0(s) = \sum_i p'_{m-i} \cdot t'_{s+i} = (p' \star t')_{m+s}$$

We can construct the sequences p' and t' in $O(n)$ time, and then compute their convolution in $O(n \log n)$ time using fast Fourier transforms.

After computing both convolutions, we can compute $\min_s(m - Both0(s) - Both1(s))$ in $O(n)$ time. The entire algorithm runs in ***$O(n \log n)$ time***. ■

Solution (powers of −1): For any integer $0 \le s \le n-m$ ("shift"), let $HD(s)$ denote the Hamming distance between $P[1..m]$ and $T[s+1..s+m]$; we need to compute $\min_s HD(s)$.

First we modify the arrays to make Hamming distance behave more like a vector dot-product. For any index i, define

$$P'[i] = \begin{cases} 1 & \text{if } P[i] = 1 \\ -1 & \text{if } P[i] = 0 \end{cases} \qquad T'[i] = \begin{cases} 1 & \text{if } T[i] = 1 \\ -1 & \text{if } T[i] = 0 \end{cases}$$

Then for any shift $0 \le s \le n-m$, we have

$$\sum_{i=1}^{m} P'[i] \cdot T'[s+i] \;=\; \sum_{i=1}^{m} \big(1 - 2\big[P[i] \ne T[s+i]\big]\big) \;=\; m - 2 \cdot HD(s)$$

Now define two sequences p and t by setting $p_i = P'[m-i]$ and $t_i = T'[i]$ for each index i. Then for all s we have

$$\sum_{i=1}^{m} P'[i] \cdot T'[s+i] \;=\; \sum_{i} p_{m-i} \cdot t_{s+i} \;=\; (p \star t)_{m+s}$$

and thus $HD(s) = (m - (p \star t)_{m+s})/2$.

We can construct the sequences p and t in $O(n)$ time, compute their convolution in $O(n \log n)$ time using fast Fourier transforms, and finally compute $\max_s (m - (p \star t)_{m+s})/2$ in $O(n)$ time. The entire algorithm runs in $\boldsymbol{O(n \log n)}$ ***time***. ■

Solution (squared differences): For any integer $0 \le s \le n-m$ ("shift"), let $HD(s)$ denote the Hamming distance between $P[1..m]$ and $T[s+1..s+m]$; we need to compute $\min_s HD(s)$.

We can express the Hamming distance $HD(s)$ as follows:

$$\begin{aligned} HD(s) &= \sum_{i=1}^{m} (P[i] - T[i+s])^2 \\ &= \underbrace{\sum_{i=1}^{m} P[i]^2}_{SumP} \;-\; 2 \cdot \underbrace{\sum_{i=1}^{m} P[i] \cdot T[i+s]}_{Both1(s)} \;+\; \underbrace{\sum_{i=1}^{s} T[i+s]^2}_{SumT(s)} \end{aligned}$$

(We can remove the squaring because $0^2 = 0$ and $1^2 = 1$!) We compute each of the terms $SumP$, $Both1(s)$, and $SumT(s)$ for all s as follows:

- We can compute $SumP$ in $O(m)$ time by brute force, once for all s.

- We can compute the third term $SumpT(s)$ for all s in $O(n)$ time using the recurrence $SumpT(s) = SumpT(s-1) - T[s] + T[s+m]$.

- Finally, we compute $Both1(s)$ for all s using convolution. Specifically, we define two sequences p and t by setting $p_i = P[m-i]$ and $t_i = T[i]$ for each index i. Then for all s we have

  $$Both1(s) \;=\; \sum_{i=1}^{m} P[i] \cdot T[s+i] \;=\; \sum_{i} p_{m-i} \cdot t_{s+i} \;=\; (p \star t)_{m+s}$$

  We can construct the sequences p and t in $O(n)$ time, and then compute their convolution in $O(n \log n)$ time using fast Fourier transforms.

Finally, we compute $\max_s (SumP - 2 \cdot Both1(s) + SumT(s))$ in $O(n)$ time by brute force. The entire algorithm runs in $\boldsymbol{O(n \log n)}$ ***time***. ■

Rubric: 10 points. These are not the only correct solutions.

- 1 for using FFT/convolution at all
- 2 for correctly dealing with both 0s and 1s (separately considering 00 and 11 matches, separately considering 01 and 10 matches, mapping $(0, 1)$ to $(-1, 1)$, squared differences, etc.)
- 3 for correctly setting up convolutions (reversing either T or P)
- 3 for correctly reading the minimum Hamming distance from the convolution(s)
- 1 for time analysis (if the algorithm is mostly correct)

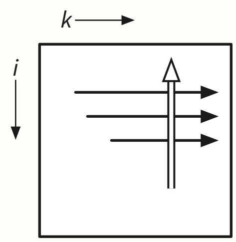
A correct algorithm that runs in $O(mn)$ or $O(mn \log n)$ time is worth at most 4 points.

CS/Math 473 ✧ Fall 2022 Midterm 1 Solutions Problem 4	Name:

Describe and analyze an algorithm to compute, given a sequence of integers separated by @ (average) signs, the ***smallest*** possible value the expression can take by adding parentheses.

Solution: Let $A[1..n]$ be the input array. For any indices $i \le k$, let $MinAve(i,k)$ denote the smallest possible value that can be obtained from the interval $A[i..k]$ by adding parentheses. We need to compute $MinAve(1,n)$. This function satisfies the following recurrence:

$$MinAve(i,k) = \begin{cases} A[i] & \text{if } i = k \\ \min\left\{ MinAve(i,j) \mathbin{@} MinAve(j+1,k) \;\middle|\; i \le j < k \right\} & \text{otherwise} \end{cases}$$



The resulting dynamic programming algorithm runs in $\boldsymbol{O(n^3)}$ ***time***. ■

Rubric: 10 points: standard dynamic programming rubric. This is not the only correct evaluation order.

The memoization arrays don't appear to have the right structure for a monotonicity speedup via SMAWK. As far as I know, this is the fastest algorithm for this problem (up to logarithmic factors).

Non-solution: Consider the following greedy algorithm: Merge the adjacent pair of numbers with the largest average (breaking ties arbitrarily), replace them with their average, and recurse. For example:

$$\underline{8 @ 6} @ 7 @ 5 @ 3 @ 0 @ 9$$
$$\underline{7 @ 7} @ 5 @ 3 @ 0 @ 9$$
$$\underline{7 @ 5} @ 3 @ 0 @ 9$$
$$6 @ 3 @ \underline{0 @ 9}$$
$$\underline{6 @ 3} @ 4.5$$
$$\underline{4.5 @ 4.5}$$
$$4.5$$

With the right data structures, this algorithm can be implemented to run in $O(n \log n)$ time; the only real bottleneck is maintaining a priority queue of adjacent pairs.

Unfortunately, this greedy algorithm does ***not*** always compute the optimal expression. Consider the input $2 @ 5 @ 0 @ 6$. The greedy algorithm outputs $(2 @ 5) @ (0 @ 6) = 3.25$, but the optimal expression is $2 @ (5 @ (0 @ 6)) = 3$. ♣