1. Design an algorithm that returns *the number of* edges between two given subsets of vertices, using only calls to AnyFriendsBetween.

Solution (divide and conquer): For any subsets S and T of vertices of G, let $m(S,T)$ denote the number of edges between S and T. Suppose we partition S into two disjoint subsets S_1 and S_2 and similarly partition T into two disjoint subsets T_1 and T_2. We immediately have

$$m(S,T) = m(S_1,T_1) + m(S_1,T_2) + m(S_2,T_1) + m(S_2,T_2). \tag{1}$$

This simple observation leads to the following natural recursive edge-counting algorithm.

In the general case, we partition S into two subsets S_1 and S_2 *whose sizes differ by at most* 1; without loss of generality, $|S_1| = \lfloor |S|/2 \rfloor$ and $|S_2| = \lceil |S|/2 \rceil$. We similarly partition T into two subsets T_1 and T_2 whose sizes differ by at most 1. Then for each $i, j \in \{1,2\}$, we call AnyFriendsBetween(S_i, T_j); if this call returns True, we recursively count the edges between S_i and T_j. Finally, we return the sum of the results of all recursive calls.

There are two base cases to consider. If either S or T are empty, there are obviously zero edges between them. If S and T each contain a single vertex, our recursive divide-and-conquer strategy could fall into an infinite loop; instead we call AnyFriendsBetween(S,T) to determine if those two vertices are connected. In all other cases, each recursive call is passed fewer vertices than its parent call.

```
CountFriendsBetween(S, T):
  if S = ∅ or T = ∅
      return 0
  if |S| = 1 and |T| = 1
      return AnyFriendsBetween(S, T)
  arbitrarily partition S into two halves S1 and S2
  arbitrarily partition T into two halves T1 and T2
  for i ← 1 to 2
      for j ← 1 to 2
          if AnyFriendsBetween(Si, Tj)
              m(Si, Tj) = CountFriendsBetween(Si, Tj)
          else
              m(Si, Tj) = 0
  return m(S1, T1) + m(S2, T1) + m(S1, T2) + m(S2, T2)
```

Correctness follows by induction from Equation (1).

Time analysis: Let n denote the the total number of input vertices ($n = |S| + |T|$), and let $m = m(S,T)$. We analyze the "running time" of our algorithm in terms of these two parameters.

Each node in the recursion tree for CountFriendsBetween(S,T) corresponds to a recursive call CountFriendsBetween(S',T') for some subsets $S' \subseteq S$ and $T' \subseteq T$.

Each recursive call invokes the subroutine AnyFriendsBetween at most four times. So up to constant factors, the running time of our algorithm is at most the number of nodes in the recursion tree.

Ignoring rounding, the total number of vertices passed to each recursive call is at most half the number passed to its parent. it follows that the recursion tree has depth $O(\log n)$.

We can classify the nodes of the recursion tree into three types:

(i) *Edge leaves.* These correspond to recursive calls CountFriendsBetween(S', T') where $|S'| = |T'| = 1$ and AnyFriendsBetween$(S', T') =$ True. Because each edge between S and T is counted exactly once, there are exactly m edge leaves.

(ii) *Internal nodes.* These correspond to recursive calls that make further recursive calls. Every internal node is an ancestor of at least one edge leaf, and every edge leaf has at most $O(\log n)$ ancestors. It follows that there are at most $O(m \log n)$ internal nodes.

(iii) *Empty leaves.* These correspond to recursive calls that return 0. (In particular, this includes recursive calls where either S' or T' is empty.) Every empty leaf is a child of an internal node, and every internal node has at most four children. It follows that there are at most $O(m \log n)$ empty leaves.

We conclude that our algorithm calls AnyFriendsBetween at most $\boldsymbol{O(m \log n)}$ ***times***. In particular, if m is significantly less than n^2, our algorithm runs in subquadratic "time". ■

More refined analysis implies the tight worst-case upper bound $O(m \log(n^2/m))$. If we allow the algorithm to use randomness (as we shall see later in the course), then one can compute a $(1+\varepsilon)$-approximation of the number of edges using only $O(\text{polylog}\, n)$ calls to the subroutine (where the big-O constant depends on ε). This algorithm is exponentially faster but substantially more complex. See the following paper if you are interested:

[1] Paul Beame, Sariel Har-Peled, Sivaramakrishnan Natarajan Ramamoorthy, Cyrus Rashtchian, Makrand Sinha. Edge estimation with independent set oracles. *ACM Transactions on Algorithms (TALG)* 16(4):1-27, 2020.

Rubric: 10 points = 3 for the algorithm + 7 for the time analysis (**not** the other way around). This may not be the only correct solution; this is not the only correct analysis of this algorithm. A brute-force algorithm that runs in $O(n^2)$ time, with correct analysis, is worth at most 4 points. (For example: consider every pair of vertices; and invoke the subroutine on those singleton sets.).

A correct divide and conquer algorithm that provides **some** justification of why it must be faster than brute-force is worth 8 points. Note that the recurrence relation $T(n, n) = 4T(n/2, n/2) + O(1)$ for the divide and conquer algorithm only implies an $O(n^2)$ time bound.

2. This question asks you to design and analyze efficient algorithms to compute the number of *distinct* interval sums in a given array $A[1..n]$.

   (a) Describe an algorithm that runs in $O(n^2 \log n)$ time.

Solution: First we compute a two-dimensional array $S[1..n, 1..n]$, where $S[i, j]$ contains the sum of all entries in the interval $A[i..j]$, in $O(n^2)$ time:

```
for i ← 1 to n
    S[i, i] ← A[i]
    for j ← i + 1 to n
        S[i, j] ← S[i, j − 1] + A[j]
```

Then we count the number of distinct entries in S (on or above the main diagonal), either by sorting and removing duplicates, inserting them into a balanced binary search tree, or inserting them into a hash table. The first two algorithms run in $\boldsymbol{O(n^2 \log n)}$ ***time***; the hashing-based algorithm runs in $\boldsymbol{O(n^2)}$ ***expected time***. ■

Solution (prefix sums): First we compute an array $P[0..n]$ of *prefix* sums, where $P[i]$ is the sum of the first i entries in the input array A.

```
P[0] ← 0
for i ← 1 to n
    P[i] ← P[i − 1] + A[i]
```

Then for any indices $i \le j$, the sum of numbers in the interval $A[i..j]$ is exactly $P[j] - P[i-1]$. So it remains to count the number of distinct differences $P[j] - P[i]$. We can record all such differences (with duplicates) into a single array by brute force in $O(n^2)$ time, and then remove duplicates by sorting and scanning in $\boldsymbol{O(n^2 \log n)}$ ***time***. ■

Rubric: 4 points = 3 for algorithm + 1 for analysis. −½for using hashing without the word "expected" in the time bound.

(b) Describe an algorithm that runs in $O(M \log M)$ time, where $M = \sum_i A[i]$.

Solution: Similarly to our second solution to part (a), we start by by computing all prefix sums in A; however, instead of a simple array of prefix sums, we compute an array $P[0..M]$ of bits, where $P[t] = 1$ if and only if some prefix of A sums to t.

```
PrefixBits(A[1..n]):
  ⟨⟨Compute sum of all entries⟩⟩
  sum ← 0
  for i ← 1 to n
      sum ← sum + A[i]
  ⟨⟨Allocate and clear bit array⟩⟩
  allocate P[0..sum]
  for t ← 1 to sum − 1
      P[t] ← 0
  ⟨⟨Set bits for prefix sums⟩⟩
  P[sum] ← 1
  for i ← n down to n
      sum ← sum − A[i]
      P[sum] ← 1
  return P
```

Next we compute a second array $\overline{P}[0..M]$ by reversing P, so that $\overline{P}[i] = P[M-i]$.

We compute the convolution $Q[0..2M] = P * \overline{P}$ in $O(M \log M)$ time using fast Fourier transforms. This convolution array is a palindrome; for any integer k, both entries $Q[M+k] = Q[M-k]$ store the number of intervals in the input array whose sum is equal to k.

Finally, we count and return the non-zero entries in $Q[0..M-1]$ (or equivalently, in $Q[M+1..2M]$.

Because the input array A contains only positive integers, we must have $M > n$, so creating the initial bit array P takes $O(n+M) = O(M)$ time. The rest of the algorithm is dominated by the convolution step, which takes ***O(M log M) time***, as required. ■

Rubric: 6 points = 1 for using FFTs at all + 2 for setting up the correct convolution + 2 for other algorithm details + 1 for time analysis (but only if the algorithm is mostly correct)

3. 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

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

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 ***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:

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

(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 $SumpT(s) = SumpT(s-1) - T[s]^2 + T[s+m]^2$.
- 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 ***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.