This document gives model solutions to the assignment problems. Note that alternative solutions may exist.

Please note that for greedy algorithms we expect a higher standard of proof and justification that your algorithm is correct. More explicitly, this means that a higher percentage of the total marks will be allocated to your proofs and explanations compared to other assignments, and about 10% of the total marks will be specifically allocated towards the clarity and conciseness of your explanations.

Question 1

[30 marks] Antoni is playing a game with some blocks. He starts with a line of n stacks of blocks with heights $h_1, \ldots, h_n$, and in a single move is allowed to split any stack into two (thus increasing the number of stacks by 1). Stacks can only be split *in-place*. That is, when splitting stack h_i, the two resulting stacks become h_i and h_{i+1}. Antoni wins the game by finishing with a sequence of stacks that is in non-decreasing order.

1.1 [7 marks] Show that it is always possible to win the game.

Reduce each block of height h_i to h_i blocks of height 1. This is guaranteed to be non-decreasing, because all blocks will have a height of 1.

1.2 [23 marks] Write an algorithm that finds the minimum number of moves required to win. Your algorithm should run in $O(n)$ time.

You are only required to **count** the minimum number of moves. You do not need to actually perform them.

Notation

It is helpful to bundle splitting operations together, for example splitting a stack h_i into multiple stacks in one operation.

After splitting the stack with index i, we shall denote the number of new stacks as k_i, and we shall arrange them in non-decreasing order. We shall denote the height of the first (smallest) new stack as low_i, and the height of the last (largest) new stack as $high_i$. If a stack is not split, then $low_i = h_i$ and $high_i = h_i$.

For example, lets say that stack 2 of height of $h_2 = 5$. If we split it into stacks of size 3 and 2, we shall arrange it as $2, 3$. So

$$\begin{aligned} low_2 &= 2 \\ high_2 &= 3 \\ k_2 &= 2 \end{aligned}$$

Present a greedy solution.

Then, we start from index $n-1$, and work towards index 1. For each stack h_i, we want to

A) Split the stack into k_i stacks such that $high_i \leq low_{i+1}$, and minimise the number of new stacks.

B) Split the stacks in such a way that low_i is maximised.

Note that we do not split stack h_n, because of objective b - we want to maximise low_i. Hence $low_n = h_n, high_n = h_n, k_n = 1$.

It is always true that $h_i \leq k_i \times low_{i+1}$, because $h_i = \text{sum of new stack heights} \leq k_i \times high_i \leq k_i \times low_{i+1}$. Since we want to minimise the number of new stacks we set

$$k_i = \left\lceil \frac{h_i}{low_{i+1}} \right\rceil.$$

Then, low_i is maximised when all stacks are as balanced as possible, so

$$low_i = \left\lfloor \frac{h_i}{k} \right\rfloor = \left\lfloor \frac{h_i}{\left\lceil \frac{h_i}{low_{i+1}} \right\rceil} \right\rfloor.$$

Our answer is the total number of splits, which is the difference between the number of new stacks and the number of initial number of stacks.

$$\left(\sum_{i=1}^{n} k_i \right) - n$$

This can be computed by keeping a tally of the total number of splits as you traverse the stacks.

Proof of correctness.

For the sake of contradiction we shall disobey objective B, that is, to have low_i to be less than the maximum possible low_i. Because the new stack heights have to be non-decreasing, $low_1 \leq high_1 \leq low_2 \leq high_2 \leq \cdots \leq low_i$, so a smaller low_i poses an unnecessary restriction on all stacks to the left.

Hence it is always optimal to obey objective B.

For the sake of contradiction we shall disobey objective A, that is, to have a k_i that is larger than necessary. From previously, we know that $h_i \leq k_i \times low_{i+1}$, so $low_i \geq \frac{h_{i+1}}{k_i}$. When k_i is larger, low_i will either stay the same or become smaller. From our previous argument, it is clear that a smaller low_i cannot lead to a more optimal solution.

Hence it is always optimal to obey objective A.

Time complexity analysis.

We look at each index once. Also note that each k_i and each low_i can be computed in constant time. Hence we have a total time complexity of $O(n)$.

Question 2

[30 marks] UNSW is competing in a programming varsity competition, and $2k$ students have registered. This competition requires students to compete in pairs, and so UNSW has run a mock competition to help determine the pairings. The results for each of the $2k$ students have been stored in an array L, and each result is a non negative integer. The score of each pair is the product of the marks of the two students, and the final score of each university is the sum of all pairs' scores.

For example, if the marks of the students are $L = [1, 4, 5, 3]$, then the pairs $(1, 3)$ and $(4, 5)$ give the largest total score of $1 \cdot 3 + 4 \cdot 5 = 3 + 20 = 23$.

2.1 **[10 marks]** Suppose the marks of 4 students are a, b, c, d where $a \leq b \leq c \leq d$. Show that $ab + cd \geq ac + bd \geq ad + bc$. You can assume that all marks are non-negative integers.

This subquestion should be solved independently of the overall question's context.

We first show that $ab + cd \geq ac + bd$. Since $d \geq a$ and $c - b \geq 0$, we have that

$$\begin{aligned} & d \geq a \\ \Leftrightarrow\ & d(c-b) \geq a(c-b) \\ \Leftrightarrow\ & cd - bd \geq ac - ab \\ \Leftrightarrow\ & ab + cd \geq ac + bd. \end{aligned}$$

To show that $ac + bd \geq ad + bc$, since $b \geq a$ and $d - c \geq 0$, we have that

$$\begin{aligned} & b \geq a \\ \Leftrightarrow\ & b(d-c) \geq a(d-c) \\ \Leftrightarrow\ & bd - bc \geq ad - ac \\ \Leftrightarrow\ & ac + bd \geq ad + bc. \end{aligned}$$

We have shown that $ab + cd \geq ac + bd$ and $ac + bd \geq ad + bc$ and thus we have $ab + cd \geq ac + bd \geq ad + bc$.

2.2 **[12 marks]** Design a $O(k \log k)$ algorithm which determines the maximum score that UNSW can achieve to ensure it has the best chance of winning.

Present a greedy solution.

We first sort the array in non-increasing order with merge sort. Then, we traverse the array to pair up elements with index $2i - 1$ and index $2i$ where $i = 1...k$ and sum their product. The maximum score we can get is $\sum_{i=1}^{k} L_{2i-1} * L_{2i}$.

Proof of correctness.

Suppose that the k pairs formed using the above algorithm are $(a_1, b_1), (a_2, b_2), ..., (a_k, b_k)$, where $a_1 b_1 \leq a_2 b_2 \leq ... \leq a_k b_k$ since we pair the two largest elements each time. According to the above algorithm, we know that $a_1 \leq a_2 \leq ,, , \leq a_k$ and $b_1 \leq b_2 \leq ,, , \leq b_k$, let σ be a permutation of 1, 2, ..., n, we can prove that $a_1 b_{\sigma(1)} + a_2 b_{\sigma(2)} + ... + a_k b_{\sigma(k)} \leq a_1 b_1 + a_2 b_2 + ... + a_k b_k$ without loss of generality.

We will prove $a_1 b_{\sigma(1)} + a_2 b_{\sigma(2)} + ... + a_k b_{\sigma(k)} \leq a_1 b_1 + a_2 b_2 + ... + a_k b_k$ by induction.
Base Case: We already proved that the statement holds when n = 2 in 2.1.
Induction Hypothesis: Assume that the statement holds when n = k, $a_1 b_{\sigma(1)} + a_2 b_{\sigma(2)} + ... + a_k b_{\sigma(k)} \leq a_1 b_1 + a_2 b_2 + ... + a_k b_k$.
Induction Step: Suppose this is true for $k - 1$ and there exists a m such that $\sigma(m) = k$. Since $a_k \geq a_m$ and $b_k \geq b_{\sigma(k)}$, we have that $a_k - a_m \geq 0$ and $b_k - b_{\sigma(k)} \geq 0$, multiplying them gives $0 \leq (a_k - a_m)(b_k - b_{\sigma(k)})$. Rearranging its terms gives $a_m b_k + a_k b_{\sigma(k)} \leq a_m b_{\sigma(k)} + a_k b_k$. This also implies $a_1 b_{\sigma(1)} + ... + a_m b_{\sigma(m)} + ... + b_k b_{\sigma(k)} \leq a_1 b_{\sigma(1)} + ... + a_m b_{\sigma(k)} + .. + a_k b_k$, where $b_{\sigma(m)} = b_k$.

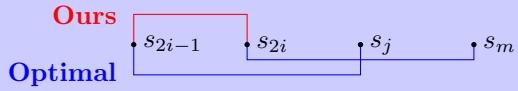
By the induction hypothesis, we know that $a_1 b_{\sigma(1)} + ... + a_m b_{\sigma(k)} + ... + a_{k-1} b_{\sigma(k-1)} \leq a_1 b_1 + ... + a_m b_m + ... + a_{k-1} b_{k-1}$, which means $a_1 b_{\sigma(1)} + ... + a_m b_{\sigma(k)} + .. + a_k b_k \leq a_1 b_1 +$

$... + a_{k-1}b_{k-1} + a_k b_k$. That is, $a_1 b_{\sigma(1)} + ... + a_m b_{\sigma(m)} + ... + b_k b_{\sigma(k)} \leq a_1 b_{\sigma(1)} + ... + a_m b_{\sigma(k)} + .. + a_k b_k \leq a_1 b_1 + ... + a_{k-1} b_{k-1} + a_k b_k$ as required.

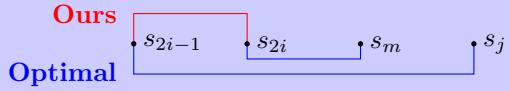
Alternate proof using exchange arguments

Let student scores be $s_1, \ldots, s_{2k}$. Consider an optimal solution which pairs elements as $(a_1, b_1), (a_2, b_2), \ldots, (a_k, b_k)$ ordered so that $a_i \leq b_i$ and $a_i \leq a_{i+1}$. Suppose that this solution differs from our solution. Then there is some left-most pair (a_i, b_i) such that $a_i = s_{2i-1}$ and $b_i = s_j$ where $j > 2i$. We have two cases, depending on whether the optimal solution pairs s_{2i} with an element between a_i and b_i, or after b_i. Our solution pairs (s_{2i-1}, s_{2i}), and the optimal solution pairs (s_{2i-1}, s_j). Let the optimal solution pair (s_{2i}, s_m). Then the two possibilities are:

[A]



[B]



[A] In this case, we may exchange pairings so (s_{2i-1}, s_{2i}) is a pair, and (s_j, s_m) is a pair. Since $s_{2i-1} \leq s_{2i} \leq s_j \leq s_m$, question (2.1) shows that

$$s_{2i-1}s_{2i} + s_j s_m \geq s_{2i-1}s_j + s_{2i}s_m,$$

so that the total sum does not decrease.

[B] In this case, we similarly exchange pairings so (s_{2i-1}, s_{2i}) is a pair, and (s_m, s_j) is a pair. By question (2.1) again, with $s_{2i-1} \leq s_{2i} \leq s_m \leq s_j$,

$$s_{2i-1}s_{2i} + s_m s_j \geq s_{2i-1}s_j + s_{2i}s_m.$$

After exchanging the pairs, the optimal solution remains optimal, and has more adjacent pairs, so is closer to our solution. Continuing this until the solution becomes ours proves that our solution must be optimal.

Time complexity analysis.

Sorting the array in non-increasing order takes $O(k \log k)$, summing and taking the product of two elements take O(1) each, performing them on each of the 2k elements takes $O(k)$. Therefore, the total time complexity is $O(k) + O(k \log k) = O(k \log k)$.

2.3 **[8 marks]** USYD is also competing in this competition and they have also run a mock competition. UNSW gets to choose USYD's team pairings. Design a $O(k \log k)$ algorithm which determines the minimum score that USYD can achieve.

Present a greedy solution.

We first sort the array in non-increasing order with merge sort. Then, we traverse the array to pair up the largest element with the smallest element, the second largest element with the second smallest element and so on, that is, index $i + 1$ and index $2k - i$ where $i = 0...k - 1$ and sum their product. The minimum score we can get is $\sum_{i=0}^{k-1} L_{i+1} * L_{2k-i}$.

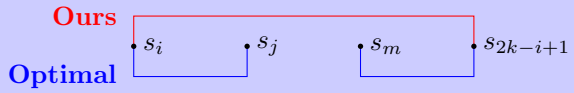
Proof of correctness.

Similar to what we did in the proof of 2.2, suppose that the k pairs formed using the above algorithm are $(a_1, b_1), (a_2, b_2), ..., (a_k, b_k)$, where every b_i for is larger than every a_i for $i = 1, ..., n$. According to the above algorithm, we know that $a_1 \leq a_2 \leq, , \leq a_k$ and $b_1 \geq b_2 \geq , , , \geq b_k$, we can prove that $a_1 b_{\sigma(1)} + a_2 b_{\sigma(2)} + ... + a_k b_{\sigma(k)} \geq a_1 b_1 + a_2 b_2 + ... + a_k b_k$ without loss of generality and apply it to the proof in 2.2 with $-b_i$ instead of b_i since negating reverses the sign.

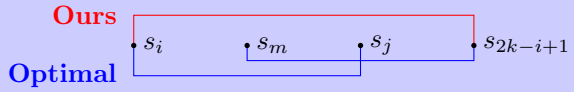
Alternate proof using exchange arguments

The proof is almost identical in structure to the proof in question (2.2). Let student scores be $s_1, \ldots, s_{2k}$. Consider an optimal solution which pairs elements as $(a_1, b_1), (a_2, b_2), \ldots, (a_k, b_k)$ ordered so that $a_i \leq b_i$ and $a_i \leq a_{i+1}$. Suppose that this solution differs from our solution. Then there is some left-most pair (a_i, b_i) such that $a_i = s_i$ and $b_i = s_j$ where $i < j < 2k-i+1$. Our solution pairs (s_i, s_{2k-i+1}), and the optimal solution pairs s_{2k-i+1} with some other element, say s_m. Note that because we ordered the pairs so that they go from outside to inside, we must have $i < m < 2k - i + 1$. This gives us the following two cases:

[A]



[B]



[A] In this case, we may exchange pairings so (s_i, s_{2k-i+1}) is a pair, and (s_j, s_m) is a pair. Since $s_i \leq s_j \leq s_m \leq s_{2k-i+1}$, question (2.1) shows that

$$s_i s_{2k-i+1} + s_j s_m \leq s_i s_j + s_m s_{2k-i+1},$$

so that the total sum does not increase.

[B] In this case, we similarly exchange pairings so (s_i, s_{2k-i+1}) is a pair, and (s_m, s_j) is a pair. By question (2.1) again, with $s_i \leq s_m \leq s_j \leq s_{2k-i+1}$,

$$s_i s_{2k-i+1} + s_m s_j \leq s_i s_j + s_m s_{2k-i+1},$$

After exchanging the pairs, the optimal solution remains optimal, and has more pairs between opposite elements in the array, so is closer to our solution. Continuing this until the solution becomes ours proves that our solution must be optimal.

Time complexity analysis.

Sorting the array in non-increasing order takes $O(k \log k)$, summing and taking the product of two elements take O(1) each, performing them on each of the 2k elements takes $O(k)$. Therefore, the total time complexity is $O(k) + O(k \log k) = O(k \log k)$.

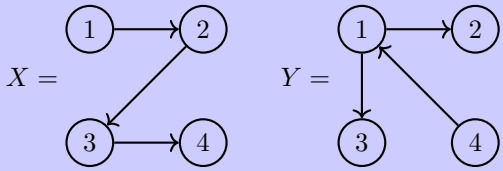
Question 3

[20 marks] Ryno needs your help! He has two directed acyclic graphs X and Y. Each of these graphs have n vertices, labelled $1, \ldots, n$. He wants to know whether there is an ordering of $[1, \ldots, n]$ which is a topological ordering of ***both*** graphs, and if so, what that ordering is.

A trivial example where this ordering does *not* exist is when $n = 2$, and X has an edge between $(1, 2)$ and Y has an edge between $(2, 1)$.

3.1 **[5 marks]** Provide another *non-trivial* example where this ordering does *not* exist. Non-trivial in this case means that an edge cannot appear in one direction in X, and in the opposite direction in Y, as per the example earlier.

One example with $n = 4$, X has edges between (1, 2), (2, 3), (3, 4) and Y has edges between (4, 1), (1, 2), (1, 3). In this case, for X, vertex 4 can only be accessed after vertex 1, while for Y, vertex 4 is before vertex 1 (doesn't require 1 to be accessed).



3.2 **[15 marks]** Design an $O(n^2)$ algorithm which solves Ryno's problem.

Present a solution.

First we create a new graph Z with vertices $[1, ..., n]$. For every edge (u, v) in X **and** Y, we insert (u, v) into Z. This graph Z is the union graph of X and Y (it contains every edge found in either X or Y). Next, perform Kahn's algorithm (the topological sort shown in lectures) on the graph Z. If Kahn's algorithm detects a cycle in Z, it means that there is no permutation of $[1, ..., n]$ that is a topological sort of both X and Y. If it successfully terminates, then the resulting topsort of Z is also a topsort of both X and Y as desired.

Proof of correctness.

We need to prove two claims to ensure that the algorithm is correct.

Claim 1: If Z does not have a cycle, then any topsort of Z is also a topsort of X and Y

Use L_Z to denote a topological ordering of Z. For every edge (u, v) in X, we have (u, v) in Z. Since L_Z is a topological ordering of Z, u must precede v in L_Z, and therefore L_Z is also a topological ordering of X. The same argument can be made for Y, and so claim 1 is true.

This means that if Kahn's algorithm terminates with a topsort of Z it must be a valid answer.

Claim 2: If there is a permutation of $[1, ..., n]$ that is a topsort of both X and Y, then it must also be a topsort of Z

Let L_Z denote the permutation that is a topsort of both X and Y. Every edge (u, v) in Z must either also be an edge in X or Y. Since L_Z is a topsort of both X and Y, then u must precede v in L_Z and it follows that L_Z is also a topsort of Z, proving claim 2.

The contrapositive of Claim 2 implies that if Z has no topsort, then there is no permutation of $[1, ..., n]$ that is a topsort of both X and Y. This means that if Kanh's algorithm detects a cycle in Z, then there is no valid ordering that can be returned.

Time complexity analysis.

Since DAGs have at most $n * (n+1)/2 = O(n^2)$ edges, constructing the graph Z takes $O(n^2)$ time. Running Kahn's algorithm to compute the topsort of Z also takes $O(V + E) = O(n^2)$ time. Clearly, the entire algorithm will terminate in worst case $O(n^2)$ time.

Question 4

[20 marks] Alice and Bob are meeting up at the park. Alice arrives at time $t \in [1, ..., n]$ with probability a_t (where $\sum_{t=1}^{n} a_t = 1$). Bob arrives at time $t \in [1, ..., n]$ with probability b_t (where

$\sum_{t=1}^{n} b_t = 1$). Assume the time t is in minutes.

4.1 **[4 marks]** Provide an expression for the probability that Alice arrives k minutes before Bob (where k can be negative).

We observe that we want to compute the probability that Alice arrives at time t and Bob arrives at time $t + k$. Since the probability that Alice and Bob arrive are independent, the probability that Alice arrives at time i and Bob arrives at time $i + k$ is the product of the probability that Alice arrives at time i and the probability that Bob arrives at time $i + k$. Therefore,

$$\begin{aligned} P(\text{Alice arrives at time } i \text{ and Bob arrives at time } i+k) &= P(\text{Alice arrives at time } i) \\ &\quad \times P(\text{Bob arrives at time } i+k) \\ &= a_i b_{i+k}, \end{aligned}$$

where $b_{i+k} = 0$ for $i + k > n$.

Therefore, we sum this over all possible times that Alice arrives to get the probability that Alice arrives k minute before Bob being

$$P_k = \sum_{i=1}^{n} a_i b_{i+k}.$$

It is also perfectly fine to interchange indices for a_i and b_j. Doing so yields

$$P_k = \sum_{i=k+1}^{n+k} a_{i-k} b_i.$$

However, observe that, for $i > n$, $b_i = 0$; therefore, it is enough to enforce the upper bound to be n. Therefore, we obtain

$$P_k = \sum_{i=k+1}^{n} a_{i-k} b_i$$

as an equivalent formulation. Note: the bounds can be from $1..n$ as well.

4.2 **[10 marks]** Design an $O(n \log n)$ algorithm that computes the probability that Alice arrives before Bob.

How might FFT fit into this problem?

Throughout the solution, assume that $k > 0$; if $k < 0$, the argument is similar with slightly different indexing. Therefore, it suffices to assume that $k > 0$.

From the first subquestion, observe that we have an expression to describe the probability that Alice arrives k minutes before Bob. To then compute the probability that Alice arrives before Bob, we essentially let k vary to obtain all possible cases for which Alice arrives before Bob.

In other words, we want to pair up a_t with all possible b_{t+k} values for each possible t.

Define $A = \langle a_1, \ldots, a_n \rangle$ and $B = \langle b_1, \ldots, b_n \rangle$ to be the sequences of Alice and Bob's probabilities respectively. We can form the corresponding polynomials $P_A(x) = a_1 x + \cdots + a_n x^n$ and $P_B(x) = b_1 x^{-1} + b_2 x^{-2} + \cdots + a_n x^{-n}$ in linear time. Note that, although P_B is not a

polynomial, we can transform P_B into a suitable polynomial by multiplying every term by x^α for a suitable α.

Notice that both P_A and P_B have n many terms; therefore, $P_C(x) = P_A(x) \cdot P_B(x)$ will have $2n$ many terms altogether. Thus, to uniquely determine the polynomial P_C, we require exactly $2n-1$ many points. To do so, we will pad $n-1$ zeroes to the end of the sequences of A and B respectively such that both sequences have precisely $2n-1$ terms; that is, we have

$$A = \langle a_1, \ldots, a_n, \underbrace{0, \ldots, 0}_{n-1} \rangle, \quad B = \langle b_1, \ldots, b_n, \underbrace{0, \ldots, 0}_{n-1} \rangle.$$

We can obtain the $2n-1$ terms by evaluating P_A and P_B at the $(2n-1)$ roots of unity. Using the Fast Fourier Transform, computing the corresponding DFTs

$$\{P_A(1), P_A(\omega_{2n-1}), \ldots P_A(\omega_{2n-1}^{2n-2})\}, \quad \{P_B(1), P_B(\omega_{2n-1}), \ldots P_B(\omega_{2n-1}^{2n-2})\}$$

can be done in $O(n \log n)$ time. We can then compute the product using element-wise multiplication in linear time to obtain

$$\{P_A(1) \cdot P_B(1), P_A(\omega_{2n-1}) \cdot P_B(\omega_{2n-1}), \ldots P_A(\omega_{2n-1}^{2n-2}) \cdot P_B(\omega_{2n-1}^{2n-2})\}.$$

Finally, using the Inverse DFT algorithm, we obtain

$$\begin{aligned} P_C(x) &= P_A(x) \cdot P_B(x) \\ &= \left(\sum_{t=1}^{n} a_t x^t \right) \left(\sum_{t=1}^{n} b_t x^{-t} \right) \\ &= \sum_{t=1}^{n} a_t b_t + \sum_{t=1}^{n} \sum_{k=1}^{t-1} a_t b_{t-k} x^k + \sum_{t=1}^{n} \sum_{k=1}^{t-1} a_t b_{t+k} x^{-k} \end{aligned}$$

in $O(n \log n)$ time.

From computing $P_C(x)$, we can see that the probability that Alice arrives *before* Bob is the sum of all coefficients for which the index is negative. To return this, notice that we can convert P_C into its sequence to obtain

$$C = \left\langle \sum_{t=0}^{j} a_t b_{j-t} \right\rangle_{j=-n}^{j=n}.$$

We extract all terms for which $j < 0$ and add each of these terms to obtain the final probability, which can be computed using $O(n)$ many summations. Thus, the overall algorithm via the Fast Fourier Transform takes $O(n \log n)$ time altogether.

Alternative solution: From the first subquestion, observe that we have an expression to describe the probability that Alice arrives k minutes before Bob. To then compute the probability that Alice arrives before Bob, we essentially let k vary to obtain all possible cases for which Alice arrives some time before Bob. However, instead of considering polynomial representations, we will consider the *prefix sum* approach.

To illustrate the intuition behind this, start by fixing some i for the arrival of Bob at time i. Now, given that Bob arrives at time i, Alice can arrive any time between 1 and $i-1$. Therefore, the probability that Alice arrives *before* Bob is

$$a_1 b_i + a_2 b_i + \cdots + a_{i-1} b_i = b_i \left(a_1 + a_2 + \cdots + a_{i-1} \right).$$

Therefore, we can define a prefix sum $y_i = a_1 + \cdots + a_i$ by noting that $y_i = y_{i-1} + a_i$ and each subsequent y_i can be computed in $O(1)$ time by storing the previous sum. With this in mind, we are now ready to construct our algorithm. Denote $y_i = a_1 + \cdots + a_i$. Then the probability that Alice arrives before Bob is

$$P'_n = \sum_{k=1}^{n} y_{k-1} b_k.$$

Note that y_k takes $O(1)$ to compute and each $y_{k-1}b_k$ also takes $O(1)$ to compute. Therefore, the overall algorithm takes $n \cdot O(1) = O(n)$ time to compute.

4.3 **[6 marks]** Design an $O(n)$ algorithm to compute the probability that Alice and Bob arrive an even number of minutes apart.

Note: this is tricky!

To understand the intuition behind the solution, it is easier to consider *what* values of k (from 4.1) we want to consider. However, we need to also consider the case where Alice arrives *after* Bob as well (note that this is exactly what negative values of k allows us to do!). To do this, we define $C(x)$ exactly as we do in 4.2; that is, define

$$C(x) = A(x) \cdot B(x) = \sum_{i=-n}^{n} c_i x^i = c_{-n}x^{-n} + \cdots + c_0 + \cdots + c_n x^n,$$

for some coefficients $c_{-n}, \ldots, c_0, \ldots c_n$. From this polynomial, it is easy to see that the coefficient of x^k is *precisely* the probability that Alice arrives k minutes before Bob. Therefore, we want to extract only the even powers. To do this, we will construct the following two equations:

$$\begin{aligned} C(1) &= c_{-n} + c_{-n+2} + \cdots + c_0 + \cdots + c_{n-2} + c_n, \\ C(-1) &= c_{-n}(-1)^n + c_{-n+2}(-1)^{-n+2} + \cdots + c_0 + \ldots c_{n-2}(-1)^{n-2} + c_n(-1)^n. \end{aligned}$$

Adding these two equations cancels out all of the odd powers because, if k is odd, then

$$c_k + c_k(-1)^k = c_k - c_k = 0.$$

Therefore, the only expressions remaining in $C(1) + C(-1)$ is

$$C(1) + C(-1) = 2 \sum_{i=-\lfloor n/2 \rfloor}^{\lfloor n/2 \rfloor} c_{2i}.$$

Therefore, the probability that Alice and Bob arrive an even number of minutes apart is $(C(1) + C(-1))/2$.

To argue the time complexity, note that we can obtain $A(1)$ and $A(-1)$ in $O(n)$ time from each a_i, and similarly we can obtain $B(1)$ and $B(-1)$ from each b_i in $O(n)$ time. We can obtain $C(1)$ and $C(-1)$ using the coefficient to value representation conversion by, again, employing $2n + 1$ many multiplications. Thus, computing $C(1)$ and $C(-1)$ can be done in $O(n)$ time, and so the probability can be computed in $O(n)$ time.

Alternative solution: If the difference in arrival time between Alice and Bob is even, then either they both arrive at an even time, or they both arrive at an odd time.

Let A_{Odd} be the probability that Alice arrives at an odd time. This can be calculated in the O(n) sum

$$A_{Odd} = \sum_{i=1}^{\lfloor n/2 \rfloor} a_{2i-1}$$

A similar method can be used to calculate A_{Even}, B_{Odd} and A_{Odd}. Our final answer is the sum of the two possibilities, both arrived at an odd time, or both arrived at an even time:

$$A_{Odd} \cdot B_{Odd} + A_{Even} \cdot B_{Even}$$

Each of the 4 sums are O(n), so the algorithm as a whole is O(n).