Solution 1.
(a)

```
1: function ALGORITHM(A, k):
2:     n ← length of A                     ▷ Assigning values to variables: O(1)
3:     B ← new array of size n − k         ▷ Assigning values to variables: O(1)
4:     for i ∈ [0 : n − k] do              ▷ Loop: O(n − k)
5:         B[i] ← 0                        ▷ Assigning values to variables: O(1)
6:         for j ∈ [0 : k] do              ▷ Loop: O(k)
7:             B[i] ← B[i] + A[i + j]      ▷ Computing and assigning values: O(1)
8:     return B
```

High-level understanding
This algorithm supplied by the question is doing one *nested* loops

1. *Loop* through every position i from index $[0]$ to $[n-k-1]$
2. *Loop* from $[j = i]$ to $[j = i + k - 1]$ Compute $\sum_{j=i}^{i+k-1} A[j]$

The inner loop from $[i]$ **to** $[i+k-1]$**: lines** $[6-7]$
To compute the $\sum_{j=i}^{i+k-1} A[j]$ takes $O(k)$ time.
The outer loop from $[0]$ **to** $[n-k-1]$**: lines** $[4-7]$
For each $\sum_{j=i}^{i+k-1} A[j]$ result, we iterate the process $n-k$ times.
The nested loop combined: lines $[4-7]$
In total, total time is $(n-k)O(k) = O(k(n-k))$.
Miscellaneous
For lines $[2-3]$ and line [8], they take $O(1)$ time combined.
Conclusion
It takes $O(k(n-k)) + O(1) = O(k(n-k))$ time

(b)
High-level understanding We reuse the sum from the *second* to the *last* element of the current iteration for the next iteration. We only calculate the sum of them once, thereby avoiding repetition and improving time efficiency.

For each iteration of $i \in [0 : n-k]$, we *carry the sum from [i+1] to [i+k-1] to the next iteration*. When calculating in the next ieteration, we **update it by subtracting the first element in the previous iteration and add the last element in the current iteration**

- Base case: when $i = 0$, we calculate the sum from the first to the last index: $B[i] = \sum_{j=i}^{i+k-1} A[j] = \sum_{j=0}^{k-1} A[j]$
- Other cases: when $i \in [1 : n-k]$, we maintain a loop invariant $S = B[i-1]$ from $i-1$ to i:
  - Subtract the previous first element $A[i-1]$ from the S

- Add the current last element $A[i+k-1]$ to the S
- Re-asign the current value of S: $B[i] = S$ such that $B[i] = B[i-1] - A[i-1] + A[i+k-1]$

1. Proof of correctness We use a **loop invariant** to prove its correctness.

Definition We define the invariant $S = B[i] = \sum_{j=i}^{i+k-1} A[j]$ as shown in the description above.

Initialisation For the base case $n = 0$, because the algorithm does not need to do anything, it is correct.

Maintenance Assuming that $S_{original} = B[i-1] = \sum_{j=i-1}^{i+k-2} A[j]$ for i, and when we increment i, $S_{original}$ becomes $S_{updated} = B[i] = \sum_{j=i}^{i+k-1} A[j]$ for i.

First we reiterate what our algorithm is doing:

$$S_{updated} = S_{original} - A[i-1] + A[i+k-1]$$

Then by our induction hypothesis that $S_{original} = B[i-1] = \sum_{j=i-1}^{i+k-2} A[j]$ for i, we have

$$S_{original} = (A[i-1] + A[i] + A[i+1] + \cdots + A[i+k-2])$$

Plugging in, we have

$$\begin{aligned} S_{updated} &= (A[i-1] + A[i] + A[i+1] + \cdots + A[i+k-2]) - A[i-1] + A[i+k-1] \\ &= A[i] + A[i+1] + A[i+2] + \cdots + A[i+k-1] \\ &= \sum_{j=i+1}^{i+k} A[j] = B[i] \end{aligned}$$

Termination Once the *cursor* $[i+k-1]$ reaches the end of the array, it stops. All the **validity** of B[i] for $i \in [0 : n-k]$ have been proved. Thus the array B is correct.

2. Time complexity analysis We maintain a *nested loop*.
For the *inner loop*:

We first only need to calculate from $[0]$ to $[k-1]$ once: $k = O(k)$ time

For the *inner loop*, we only need to add and subtract two numbers: $2 \times 1 = O(1)$ time

For the *outer loop*:

We do the *inner loop operation* for $n-k$ times. Within it, the first time is $O(k)$, and the rest $n-k-1$ times are $O(1)$ as described above.

For the *nested loop*: It takes $O(k) + (n-k-1)O(1) = O(k) + O(n-k-1)$ time. Because $k < n$, $O(k) < O(n)$ and $O(n-k-1) < O(1)$, it runs in $O(n)$ time.

In conclusion, we finallly take into account some miscellaneous operations *return* statements $O(1)$ time. it takes at most $O(n) + O(1) = O(n)$. Thus, our algorithm runs in $O(n)$ time as required.

Solution 2.
(a)
High-level understanding

The whole idea for this is to store the **cumulative sum** to speed up the calculation of the next step and to **dynamically** choose whether to use *odd* or *even* attributes. When **dequeuing**, subtract the previous result from the current result.

A more in-depth look:

we have two **sums** (*odd* and *even*) for Q_n, one that ends with Q_n, one that ends with $\frac{1}{Q_n}$. We will select between the two based on a **counter**.

Now if we dequeue **integers**, we have to subtract them. The **dequeued** integers also have a **sum** (*odd* and *even*). **A large part of our description is about how to correctly select between the two types of sum (*odd* and *even*).**

Detailed description

1. Operation We have *enqueue()*, *dequeue()*, and *seeSaw()*, three operations.

Before everything, we have two *doubly linked lists* to implement this. One for storing all the elements in the *queue* (including ones deleted), another for storing the results from *seeSaw()* to speed up operations.

For each time we have an incoming integer, we **calculate all attributes according to the** *class Node* and **update attributes in** *class LinkedList*.

enqueue()	1. First, adjust the *next* pointer of the last node to point at the new *node* 2. Then, Set the *next* point of the newly added *node* to $NULL$ 3. Next, Calculate the $OriginalValue = n$ and $InverseValue = 1/n$ **(*Inverse* means multiplicative inverse, i.e., reciprocal)**. 4. For the first *node*, $sumOdd = OriginalValue$, $sumEven = InverseValue$ 5. For all nodes following that: compute the $node.sumOdd$ and $node.sumEven$ according to the next table about *class* 6. Update the *generalCounter* according to the next table about *class*
dequeue()	1. Set the *next* pointer of the *sentinel start NULL node* to the second node, thereby *abandoning* the first node 2. update the $list.sumDequeuedOdd$ and $list.sumDequeuedEven$ according to the next table about *class* 3. Update the *dequeuedCounter* according to the next table about *class*
seeSaw()	We return the $node.sum - list.sum$. For each *sum* value, we have two choices, *odd* or *even*. To determine whether to return the *odd* or *even*, see below: · if $list.dequeuedCounter\%2 == 1$ is *odd*: ⇒return $node.sumEven - list.sumEven$; · if $list.dequeuedCounter\%2 == 0$ is *even*: ⇒return $node.sumOdd - list.sumOdd$.

2. Data structure

For each *node* of the *list*, we store the *next* pointer, its value, and the cumulative sum $sumOdd$ and $sumEven$. We also update $list.generalCounter$ and $list.dequeuedCounter$ to determine whether we choose the *odd* or *even* value of *sums*.

For the whole *list*, we store all the *nodes* as well as $sumDequeuedOdd$ and $sumDequeuedEven$ to modify the $seeSaw()$ return value according to the nodes *dequeued* previously: $node.generalCounter - list.dequeuedCounter$.

Note that in the following table, the term counter does not mean 0*-based index. The counter starts from* 1.

node	*node.next*	Pointer at the next *node*
	node.prev	Pointer at the previous *node*
	node.generalCounter	The counter of the *node* counting from the first *node* of the list (including dequeued *nodes*). We increment it every time when we add a new item
	node.OriginalValue	The original value n: e.g. 19
	node.InverseValue	The inverse value $1/n$: e.g. $\frac{1}{19}$
	node.sumOdd	If $i = 1$, it is *node.OriginalValue*; For the i-th ($i > 1$) node, ·If $generalCounter\%2 == 1$ (*odd*): ⇒We add $node.OriginalValue$ to $node.sumOdd$; ·If $generalCounter\%2 == 0$ (*even*): ⇒We add $node.InverseValue$ to $node.sumOdd$; $node.sumOdd = node.prev.sumEven + node.OriginalValue$;
	node.sumEven	If $i = 1$, it is *node.InverseValue*; For the i-th ($i > 1$) node, ·If $generalCounter\%2 == 1$ (*odd*): ⇒We add $node.InverseValue$ to $node.sumEven$; ·If $generalCounter\%2 == 0$ (*even*): ⇒We add $node.OriginalValue$ to $node.sumEven$;
LinkedList	*list.dequeuedCounter*	A counter counting all the *nodes* dequeued or removed
	list.sumDequeuedOdd	First, increment *dequeuedCounter* by 1. ·If $dequeuedCounter\%2 == 0$ (*even*): ⇒add $InverseValue = \frac{1}{n}$ to it. ·If $dequeuedCounter\%2 == 1$ (*odd*): ⇒add $OriginalValue = n$ to it;
	list.sumDequeuedEven	First, increment *dequeuedCounter* by 1 If $dequeuedCounter\%2 == 0$ (*even*)): ⇒Add $OriginalValue = n$ to it; If $dequeuedCounter\%2 == 1$ (*odd*): ⇒Add $InverseValue = \frac{1}{n}$ to it;

(b) As is introduced in the lecture, the *linkedlist* data type is a correct way to implement the *queue* ADT's features: *enqueue*, *dequeue* and so on.

Our proof has three parts: **firstly**, proving that we can successfully calculate the two *odd* and *even* sum values; **secondly**, proving that we can deal with the case where elements are *dequeued* **AND** successfully *alternate* between the "Odd" and "Even" lists in order to **return** the correct value.

Please note that for the sake of discussion, we define the first integer to be Q_0 even though it may have been removed from the queue. Q_t is the first integer of the ***remaining queue*** *after dequeuing*

1. Calculating the sum of *odd* and *even*

Aim: for each *node* Q_n, our desired result is $Q_0 + \frac{1}{Q_1} + Q_2 + \dots$ for *node.sumOdd* and $\frac{1}{Q_0} + Q_1 + \frac{1}{Q_2} + \dots$ for *node.sumEven*.

Note that the Q_0 is the first inserted integer, even though it may have been dequeued. This is a way to nicely format the proof.

Proof. **We use the *mathematical induction* method to prove this:**

We **initialise** the two **invariants** to be *sumOdd* and *sumEven*. We assert that they fulfill the **desired results** as specified above.

Base case For the base case $n = 0$ and $n = 1$, we can see that $Q_0.sumOdd = Q_0$, $Q_0.sumEven = \frac{1}{Q_0}$, and $Q_1.sumOdd = Q_0 + \frac{1}{Q_1}$, $Q_1.sumOdd = \frac{1}{Q_0} + Q_1$. Thus the base case is proven.

Induction hypothesis We assume that *sumOdd* and *sumEven* holds for node Q_{k-1}; we hypothesise that it further holds for Q_k

Induction step Because $Q_{k-1}.sumOdd = Q_0 + \frac{1}{Q_1} + Q_2 + \dots$, and $Q_k.sumEven = \frac{1}{Q_0} + Q_1 + \frac{1}{Q_2} + \dots$.

We first prove it for $Q_k.sumOdd$ when the *generalCounter* is odd: by observation, when the general counter *generalCounter* is odd, the term containing Q_n should be Q_n (*OriginalValue*). Our algorithm **simulates** exactly that. Thus, it is proven for all steps afterwards.

Proving it for $Q_k.sumOdd$: when the *generalCounter* is even: by observation, when the general counter *generalCounter* is odd, the term containing Q_n should be $\frac{1}{Q_n}$ (*InverseValue*). Our algorithm **simulates** exactly that. Thus, it is proven for all steps afterwards.

Proving it for $Q_k.sumEven$: when the *generalCounter* is odd: by observation, when the general counter *generalCounter* is odd, the term containing Q_n should be $\frac{1}{Q_n}$ (*InverseValue*). Our algorithm **simulates** exactly that. Thus, it is proven for all steps afterwards.

Proving it for $Q_k.sumEven$: when the *generalCounter* is even: by observation, when the general counter *generalCounter* is odd, the term containing Q_n should be Q_n (*OriginalValue*). Our algorithm **simulates** exactly that. Thus, it is proven for all steps afterwards. □

2. Dealing with *dequeuing* and *alternating*

Aim: proving that our algorithm still stands after *dequeuing* and **returns** $node.sum - list.sumDequeued$ depending on the parity of the *dequeuedCounter*.

(Here Q_0 is the first integer that was removed and Q_t is the first integer in the *queue* after *dequeuing*.)

Proof. Our algorithm achieves this through simulation.

We observe that

$$\begin{aligned} node.sum &= Q_0 + \frac{1}{Q_1} + Q_2 + \frac{1}{Q_3} + \cdots \\ &= (\cdots + Q_{t-2} + \frac{1}{Q_{t-1}} + Q_t + \frac{1}{Q_{t+1}} + \dots) - (\cdots + Q_{t-2} + \frac{1}{Q_{t-1}}) \end{aligned}$$

Breaking it down, we have (depending on the parity)

$$\begin{aligned} node.sum &= \cdots + Q_{t-2} + \frac{1}{Q_{t-1}} + Q_t + \frac{1}{Q_{t+1}} + \ldots \\ list.sumDequeued &= \cdots + Q_{t-2} + \frac{1}{Q_{t-1}} \end{aligned}$$

We first can observe that the parity of *node.sum* and *list.sumDequeued* must be the same. So we can have either the "Odd" equation

$$\text{“}Odd'' = node.sumOdd - list.sumDequeuedOdd$$

or the "Even" equation

$$\text{“}Even'' = node.sumEven - list.sumDequeuedEven$$

and no other possibilities.

The burden now lies within whether we should choose "Odd" or "Even".

Inside the *node.Sum* equation, we can observe that when Q_t is the first integer in the queue after removal, it **must be in its** *node.OriginalValue* **form**. We also know, **from our proof above**, that Q_t will be added to *node.sumOdd* if the generalCounter $t+1$ is odd, and to *node.sumEven* when $t+1$ is even.

From our definition, $t = list.dequeuedCounter$. Hence, they have the same parity.

Thus, our **ideal algorithm** should do: if *list.dequeuedCounter* is even, use the "Odd" equation; otherwise, use the "Even" one.

By consulting the *seeSaw()* function from the tables above, our algorithm **simulates** the process perfectly. Hence its correctness. □

(c)

Time complexity (running time):

First, the most basic *enqueue* and *dequeue* operations all take $O(1)$ by detaching and reattaching the *pointers* as shown in the lectures.

Then, our method *augments* the operations by calculating the *generalCounter*, *InverseValue*, *sumOdd*, and *sumEven* of the node.

- The *generalCounter* take $O(1)$ time to increment.

- Calculating the $InverseValue = 1/n$ takes $O(1)$ for each $en/dequeue$ operation.

- For the *sumOdd* or *sumEven*, we only add **one single** value (*original* or *inverse*) on top of the previous results based on the parity of the *generalCounter*. The checking of the *if* statements and the *adding* takes $O(1)$ time in total.

Lastly, let's consider the time complexity of $seeSaw()$.

It comprises of the if conditions checking, and the *calculating*. For the ifs, we only need $O(1)$ for each function call. For *calculating*, we already have the "source materials" ready, i.e., *node.sum* and *list.sumDequeued* depending on *odd* or *even* respectively. It just takes a brief $O(1)$ to complete the operation.

In conclusion, all our operations run in $O(1)$ constant time, proving our algorithm's superiority.

Space complexity (space of our data structure):

For each *LinkedList*, we need to store:

- $O(1)$ space: $list.dequeuedCounter$;
- $O(1)$ space: $list.sumDequeuedOdd$;
- $O(1)$ space: $list.sumDequeuedEven$;

For each *node* object, we need to store:

- $O(1)$ space: the *next* pointer;
- $O(1)$ space: the *prev* pointer;
- $O(1)$ space: the $node.generalCounter$;
- $O(1)$ space: the $OriginalValue$;
- $O(1)$ space: the $InverseValue$ after calculation;
- $O(1)$ space: the $sumOdd$ after calculation;
- $O(1)$ space: the $sumEven$ after calculation.

In conclusion they all take

$$\underbrace{O(1)}_{LinkedList} + \underbrace{nO(1)}_{node} = O(n)$$

space, resulting in $O(n)$ in total.

Solution 3.
(a) High-level understanding We implement the *stack* using two *linked lists*.

One *linked list* A for doing the normal stack operation with an extra *pointer* pointing towards S_{n-1}. Another *linked list* B is updated every time we call the *sillyProd()* function. Every *node* in this extra *list* is the $\sum_{i<j<C:S_i<S_j} S_iS_j$ where C is the index of interest.

The $\sum_{i<j<C:S_i<S_j} S_iS_j$ value is **calculated based on the previous *node***.

We return the value in the last node of the *list* B, i.e., for the *sillyProd()* function.

Detailed description For *standard operations* like *push* and *pop*, we add the new *integers* to the end of the linked list when *pushing*, and remove them when *popping*.

(The $NULL$ node should be adjusted as detailed in the lectures:

- when *pushing*, reattach the *next* pointer of the last node from *NULL* to the new node;

- when *popping*, reattach the *next* pointer of the second last node from *pointing at the last node* to *NULL*.)

However, although no changes were made to the *push* operation, ***pop* is modified as follows**. Every time we *pop* on the A_{n-1}, we will also remove the corresponding B_{n-1} from the *linked list* B (which will be introduced as below). We subsequently update the *pointer* pointing towards the A_{n-1} and B_{n-1} element for each list.

1. First time: *sillyProd()* When calling it for the first time, the second *linked list* B is completely empty. We use a nested loop where the outer is j and the inner is i.

- Outer loop $j \in [0 : n]$: We have a traversal variable S_j from S_0 to S_{n-1}.

- Inner loop $i \in [0 : j]$: We <u>first</u> have a *temp* variable. For each S_j, we compare all the *nodes* S_i $(i < j)$ **prior to** it. If $S_i < S_j$, we add the S_iS_j to *temp*. If not, we just *skip* it. **After** the traversal from 0 to j is finished, we store *temp* inside *linked list* $B[j]$. And we move the *cursor* from B_j to B_{j+1}.

2. Later times: *sillyProd()* If we have already called *sillyProd()* once, we can utilise the results from before. The problem is then divided into two cases:

- Calling *sillyProd()* right after the previous function call:

You are just calling *sillyProd()* again without any *pushing* or *popping* on the A_{n-1}. In this case, just return the value B_{n-1}.

- Calling *sillyProd()* after some *pushing, popping*:

In this case, you have two lists: A from $[0]$ to $[n-1]$ is the updated list; B from $[0]$ to $[m-1]$ because B was not re-calculated after A was updated.

If $n > m$, i.e., we *pushed* more nodes than we have *popped*: all the nodes from index $B[0]$ to $B[m-1]$ are not affected. We only update the newly added integers in $B[m]$ to $B[n-1]$ by repeating the process of **1. First time**. The outer loop is now $j \in [m : n]$ while $i \in [0 : j]$. After updating, we just return the last node of B again.

It is, however, impossible to have $n \leq m$. Because by our algorithm, we decrement m every time an existing element is removed. That means that $m \leq n$ stands for all cases. And the only case where $n = m$ can only happen if the *stack* is not modified at all, which belongs to the previous case **"right after the previous function call"**

(b) The proof is long and will be divided into three parts. We first discuss why the **inner loop** algorithm works, then the **outer loop**. Lastly, we discuss how our algorithm holds for cases where you call *sillyProd()* after first calling it.

1. Inner loop $i \in [0 : j]$ Aim: proving that for a $j = c$ where c is a constant, our algorithm has $B[j = c] = \sum_{i<C:S_i<S_C} S_i S_C$.

Proof. We prove by **loop invariant**. For $i \in [0 : j = c]$, we maitain a loop invariant k to **pass on to the next iteration**. We assert that $p = \sum_{k<i\in[0:c]:S_k<S_c} S_k S_c$.

Initialisation

If $j = c = 0$ or $j = c = 1$, our algorithm compares S_0 against S_1 and works as expected.

Maintenance

We assume that for the current iteration k, the loop invariant is $p = \sum_{k<i\in[0:c]:S_k<S_c} S_k S_c$.

Assuming that it works for k, we then show how it works for $k+1$. The key **difference** between the two is that by theory, we add $S_k S_c$ to the return value if $S_k < S_c$. Our algorithm achieves this by iterating all *nodes* i in $[0 : j+1]$ and add all products of the two if $S_i < S_{j+1}$.

Termination

When $i = c - 1$, we have successfully collected all $S_i S_c$ where $S_i < S_c$. Thus its correctness is proven. □

2. Outer loop j: Calling for the first time

Aim: proving that our algorithm successfully **sum up** all the results from the **inner loop** to yield $\sum_{i<j\leq n-1:S_i<S_j} S_i S_j$.

Proof. **We prove by mathematical induction:**

Base case

For $j = 1$, i.e., a two-integer *stack*, the expected return value is $\sum_{i<j\leq 1:S_i<S_j} S_i S_j = S_0 < S_1$ if $S_0 < S_1$. Since there is only one sum to add, our algorithm proves to be correct.

Induction hypothesis

Assuming that our algorithm holds for $n - 1 = C$ (C is a constant), we intend to prove that it works for $n = C + 1$.

Inductive step

The difference between the two is that

$$\sum_{i<j<n:S_i<S_j} S_i S_j = \sum_{i<j<n-1:S_i<S_j} S_i S_j + \sum_{i<n:S_i<S_n} S_i S_n$$

Our algorithm makes up this difference by adding the **sum of product** to the **the sum of sum of product of the previous node** to obtain the <u>current</u> **sum of sum of products**. □

3. Calling for the second time Aim: proving that after calling *sillyProd()*, no matter how we modify (*push* and *pop*) the **stack**, our algorithm will still return the correct result of $\sum_{i<j\leq n-1:S_i<S_j} S_i S_j$ where n is the modified *length* of teh *stack*.

We prove by mathematical induction:

Base case We call *sillyProd()* immediately after the first call. In this case, our algorithm did not modify anything and return the B_{j-1} straight away, thus is correct.

Induction hypothesis Assuming that after **one single** modification (*popping* or *pushing* only one element), our algorithm is still correct.

Inductive step We call *sillyProd()* again after calling it once and do **only one modification (*push* or *pop*)**. In this case, the difference between the previous and current one is one node. If one node was *popped*, then by our algorithm the *linked list B* will remove the **last node**. Then we add the $\sum_{i<n-1:S_i<S_{n-1}} S_i S_{n-1}$ to B_{n-1} to obtain B_n. After that we return B_n. Thus achieving:

$$B_n = \sum_{i<j<n-2} S_i S_j + \sum_{i<n-1} S_i S_{n-1} = B_{n-1} + \sum_{i<n-1} S_i S_{n-1} = \sum_{i<j<n-1} S_i S_j$$

(c)

Time complexity:

1. Standard operations all run in $O(1)$ time as proved in the lecture. As we did not modify these two operations, the analysis is still correct.

2. sillyProd() Function

2.1 Calling for the first time: Nested loop only

After *pushing m* times and *popping n* times:

For the **inner loop** running in $i \in [0 : j]$: $O(j)$. To upperbound, assume all nodes $S_i < S_j$, thus comparing and multiplying for j time each. In total, it is $2j = O(j)$

For the **outer loop** running in $j \in [1 : n-1]$: we run this process for $n - 1 = O(n-1)$

For the **nested loop** in total, because $j < n$:

$$\sum_{j=1}^{n-2} O(j) = O(n^2)$$

2.2 Calling after modification

We have m_1 *pushings*, n_1 *poppings* and t_1 total elements, after which 1 *sillyProd()* and more m_2 *pushings* and n_2 *poppings*,..., before Calling for the *sillyProd()* this time. So the time consumed prior to this is $\sum_{i=1}^{k}(m_k + n_k) + k$; the elements left is t_k.

We then use **mathematical induction** and only consider case where we have already called *sillyProd()* $k-1$ times and modified the *stack* k times. Now we are calling it for the k-th time. As stated above, the number of integers are t_{k-1} and t_k for the two cases.

If ultimately $t_k > t_{k-1}$, then we have to make up the $diff = t_k - t_{k-1}$. We call the **inner loop** operation for $diff$ times to calculate the **sum of the sum of the products** of the newly added integers.

To upperbound, assuming again all numbers satisfy $m < n : S_m < S_n$, the operations outlined above would take **at most**

$$\sum_{j=t_{k-1}-1}^{t_k-1} 2j = O(t_k)$$

because $j, t_{k-1} < t_k$.

If ultimately $t_k < t_{k-1}$, then we can simply return B_{t_k} value which takes $O(1)$ time.

In the end, through iterating the process, the time for each **single** *sillyProd()* operations will always be $O(t_k^2) = O(n^2)$

2.3 Amortisation analysis

Since the amortised time is

$$\frac{\textbf{Number of steps (Number of steps)}}{\textbf{Number of operations}}$$

, such an upperbound analysis must maximise the numerator and minimize the denominator.

To upperbound the time, $t_k > t_{k-1}$ must be satisfied. Otherwise, to call the *sillyProd()* function will only take $O(1)$ time as proved above rather than $O(n^2)$ time. That rendered the time complexity analysis trivial.

Thus, from $k-1$ to k, the time for *sillyProd()* only would be

$$\text{Time (Number of steps)} = O(n^2)$$

as proved above in the **2.2 Calling after modification** section.

To lowerbound the number of operations, we first consider the general formula for **number of operations**:

- *pushing* or *popping*: $\sum_{i=1}^{m}(m_i + n_i)$

$$O(\text{Number of operations}) = \overbrace{\sum_{i=1}^{k}(m_i + n_i)}^{\textit{pushing} \text{ or } \textit{popping elements}} + k$$

We consider the fact that $\sum_{i=1}^{m}(m_i - n_i) = \sum_{i=1} m_i - \sum_{i=1} n_i = n$ by the definition of m times of*pushing* and n times of *popping*, and also $\sum_{i=1} n_i < n$. Thus,

$$\sum_{i=1}(m_i + n_i) = \sum_{i=1} m_i + \sum_{i=1} n_i = \sum_{i=1} m_i + (\sum_{i=1} m_i - n) = 2\sum_{i=1} m_i - n$$

To lowerbound this, because if $\sum_{i=1} n_i$ goes up, then $\sum_{i=1} m_i = n + \sum_{i=1} n_i$ will increase. To make $\sum_{i=1} m_i$ as low as possible, $\sum_{i=1} n_i = 0$. Thus $\Omega(\sum_{i=1} m_i) = n$.

In the end,

$$\Omega(\text{Number of operations}) = \Omega(n) + k = \Omega(n)$$

Combining the numerator and the denominator together,

$$O(\textit{sillyProd()}) = \frac{O(n^2)}{\Omega(n)} = O(n)$$

as required.

Space complexity: For all *pushing* and *popping*, each *node* in the *linked list* A only takes up:

- value: $O(1)$
- *next* pointer: $O(1)$

For the A, we have a extra pointer pointing at the most recent integer: $O(1)$

For the *linked list B*, for each *node*, we store only one value about the $\sum_{i<j\leq k:S_i<S_j} S_iS_C$ by adding the $\sum_{i<S_k:S_i<S_k} S_iS_k$ to the previous *node's value*.

In total, it takes

$$nO(1) + nO(1) + O(1) + nO(1) = O(n)$$

space as required.