ESO207 – Theoretical Assignment #1

Name: Sumath Rengasami V

Roll No: 241054

Date: September 3, 2025

Question 1: Maximum Product Sum

(a) Algorithm & Pseudocode

We are given two sequences $a = [a_1, \ldots, a_n]$ and $b = [b_1, \ldots, b_n]$. We want

$$P \;=\; \max_{\pi} \sum_{i=1}^{n} a_i \cdot b_{\pi(i)}.$$

The maximum is attained by pairing the sorted array: sort a in increasing order and b in increasing order, then take the dot product.

Algorithm 1 MaximumDotProduct(a,b)

```
sort a in increasing order
sort b in increasing order
P ← 0
for i = 1 to n do
    P ← P + a[i − 1] · b[i − 1]
end for
return P
```

Time Complexity: Sorting dominates: $O(n \log n)$.

(b) Handling q updates and reporting P after each

Each update is of the form (o, x):

- if $o = 1$: increment a_x by 1,
- if $o = 2$: increment b_x by 1,

and after each update we must output the maximum dot product P.

First we calculate P initially. We maintain two additional arrays: *sorted_a* and *sorted_b*, which are sorted versions of a and b (in increasing order).

For each query:

- If $o = 1$, update a
  1. Find the current value $a[x]$ and compute $a[x] \leftarrow a[x] + 1$.
  2. Locate the old value of $a[x]$ in *sorted_a* using binary search.
  3. Replace it with the new value.
  4. Add B[i] to P.
- If $o = 2$
  1. We do the same as above but with b and *sorted_b*.

After each update, the arrays *sorted_a* and *sorted_b* remain valid sorted versions of a and b. We then compute the maximum dot product using the precomputed value of the intital dot product.

Algorithm 2 Binary Search $(array, value, low, high)$

```
function BINARY_SEARCH(array, value, low, high)
    result ← −1
    while low ≤ high do
        mid ← ⌊(low + high)/2⌋
        if array[mid] = value then
            result ← mid
            low ← mid + 1
        else if array[mid] < value then
            low ← mid + 1
        else
            high ← mid − 1
        end if
    end while
    return result
end function
                ▷ binary_search returns the index of last occurrence of value in array
```

The above binary search will be used in the following algorithm.

Algorithm 3 MaximumDotProductAfterQuery(a, b, query)

```
P ← 0
sorted_a ← sorted(a)
sorted_b ← sorted(b)
for i = 1 to n do
    P ← P + sorted_a[i − 1] · sorted_b[i − 1]
end for
for all (o, x) ∈ query do
    if o = 1 then
        i ← binary_search(sorted_a, a[x − 1], 0, n − 1)
        a[x − 1] ← a[x − 1] + 1
        sorted_a[i] ← sorted_a[i] + 1
        P ← P + sorted_b[i]
    else if o = 2 then
        i ← binary_search(sorted_b, b[x − 1], 0, n − 1)
        b[x − 1] ← b[x − 1] + 1
        sorted_b[i] ← sorted_b[i] + 1
        P ← P + sorted_a[i]
    end if
    return P
end for
```

(C) Proof of Correctness

Preliminaries (Inductive Exchange Argument). Let $x_1 \le \cdots \le x_n$ and $y_1 \le \cdots \le y_n$. We prove by induction on n that the dot product $\sum_{i=1}^{n} x_i y_{\pi(i)}$ is maximized when the pairing is order-preserving, i.e., when $\pi(i) = i$ for all i.

Base case ($n = 1$). Trivial.

Inductive hypothesis. Assume the claim holds for all sizes up to k: for any nondecreasing sequences of length k, the maximum dot product is achieved by pairing equal indices.

Inductive step ($k \to k{+}1$). Consider nondecreasing sequences $x_1 \le \cdots \le x_{k+1}$ and $y_1 \le \cdots \le y_{k+1}$. Let Π be an optimal pairing for these $k{+}1$ elements. In Π, let x_{k+1} be paired with y_j.

If $j = k{+}1$, then remove this pair; by the inductive hypothesis, the remaining k pairs are optimally matched in order, so Π is the order-preserving pairing overall.

If $j < k{+}1$, then in Π some x_i (with $i \le k$) is paired with y_{k+1}. Consider the pairing Π' obtained by *swapping* these two partners, so that x_{k+1} is paired with y_{k+1} and x_i is paired with y_j, leaving all other pairs unchanged. The change in total sum is

$$\Delta \;=\; \big(x_{k+1}y_{k+1} + x_i y_j\big) \;-\; \big(x_{k+1}y_j + x_i y_{k+1}\big) \;=\; (x_{k+1} - x_i)\,(y_{k+1} - y_j) \;\ge\; 0,$$

because $x_{k+1} \ge x_i$ and $y_{k+1} \ge y_j$. Hence Π' is also optimal (and no worse than Π) and now pairs the largest elements together: $x_{k+1} \leftrightarrow y_{k+1}$.

Remove this maximal pair. By the inductive hypothesis, among the remaining k elements the maximum is achieved by pairing in order $(x_1, y_1), \ldots, (x_k, y_k)$. Therefore, for $k{+}1$ elements, the order-preserving pairing maximizes the dot product.

Conclusion. By induction, for all n the maximum $\max_\pi \sum_{i=1}^{n} x_i y_{\pi(i)}$ is achieved by the sorted, aligned pairing. In particular, if `sorted_a` and `sorted_b` are the nondecreasing sorts of a and b, then

$$P^\star(a, b) = \sum_{i=1}^{n} \texttt{sorted\_a}[i] \cdot \texttt{sorted\_b}[i].$$

Algorithm Invariants. After any iteration of the outer loop over queries, the algorithm maintains:

- **I1.** `sorted_a` is the nondecreasing sort of the current a, and `sorted_b` is the nondecreasing sort of the current b.
- **I2.** $P \;=\; \sum_{i=1}^{n} \texttt{sorted\_a}[i] \cdot \texttt{sorted\_b}[i]$.

Initialization. The algorithm set `sorted_a` and `sorted_b` to sorted copies and compute P as their aligned dot product. By the induction above, $P = P^\star(a, b)$.

Update step for a query on a (case $o = 1$). Let v be the current value of $a[x{-}1]$. The algorithm finds

$$i \leftarrow \texttt{binary\_search(sorted\_a, } v\texttt{)}$$

returning the index of the ***last occurrence*** *of* v in `sorted_a`. We then remove its contribution at position i from P , increase both $a[x{-}1]$ and `sorted_a[i]` by 1.

Why at most one adjacent swap suffices. Because i is the *last* index with value v, we have either $i = n$ or $\texttt{sorted\_a}[i{+}1] > v$. After increment, the updated value is $v{+}1$. Thus:

$$\texttt{sorted\_a}[i] \;=\; v{+}1 \;\le\; \texttt{sorted\_a}[i{+}1] \quad \text{whenever} \quad \texttt{sorted\_a}[i{+}1] \ge v{+}1,$$

so the order is already nondecreasing and no swap is needed.

Maintaining P exactly. Before modifying sorted_a[i], we subtract its paired contribution $\texttt{sorted\_a}[i] \cdot \texttt{sorted\_b}[i]$ from P. After increasing the value by 1 we simply add back the new paired term at index i, so P again equals $\sum_k \texttt{sorted\_a}[k]\texttt{sorted\_b}[k]$.

Update step for a query on b (case $o = 2$). This is symmetric to the above: we locate the last occurrence of $b[x{-}1]$ in sorted_b, increment it by 1.

Inductive Conclusion. By initialization and preservation of $I1$–$I2$ at each query, after processing every query we have that sorted_a and sorted_b are the sorted versions of the current arrays a and b, and

$$P \;=\; \sum_{i=1}^{n} \texttt{sorted\_a}[i] \cdot \texttt{sorted\_b}[i] \;=\; P^{\star}(a, b)$$

by the induction. Hence the algorithm always returns the *maximum* dot product after each update, proving correctness.

Time Complexity. Sorting the two arrays requires $O(n \log n)$ time, and computing the initial value of P takes $O(n)$. For each query, we spend $O(\log n)$ time to locate the updated element via binary search and $O(1)$ time to adjust the necessary values. Hence, the overall time complexity is

$$O(n \log n) + O(n) + q \cdot (O(\log n) + O(1)) = O\big((n + q) \log n\big).$$

Question 2: Game Score Maximization

We are given n positive integers $\texttt{nums}[1..n]$, initial score $S = 1$, and k coins. With each coin:

1. Choose an interval $[L, R]$ not chosen previously.
2. Let x be the element in $[L, R]$ with the *highest number of prime factors*; ties break by smallest index.
3. Multiply $S \leftarrow S \cdot x$.

(a) Algorithm

We begin by computing the prime factor multiplicity $\Omega(x)$ for every array element using a smallest prime factor (SPF) sieve.

For each element a_i, we identify the set of intervals where it is the chosen element: it must be the leftmost element with the maximum Ω value in that interval. To determine this region, we use two stacks:

- From the left, we find the nearest index $L_i^\star$ with $\Omega \geq \Omega(a_i)$.
- From the right, we find the nearest index $R_i^\star$ with $\Omega > \Omega(a_i)$.

The product

$$\text{cap}_i = (i - L_i^\star)(R_i^\star - i)$$

gives the number of valid intervals where a_i dominates.

Next, we sort all elements together with their capacities cap_i in decreasing order of a_i. We then distribute the k coins across elements in this order, up to each element's capacity. Formally, for each i we take

$$t = \min\{\text{cap}_i,\, k\},$$

multiply the score by a_i^t, and update $k \leftarrow k - t$. We stop once $k = 0$. This ensures that every coin is spent on the largest available element under the rules.

Algorithm 4 MaximizeGameScore($a[1..n]$, k)

$M \leftarrow \max(a[1..n])$
$\texttt{SPF} \leftarrow \textsc{SmallestPrimeFactorSieve}(M)$
for $i \leftarrow 0$ to $n-1$ **do**
    $\Omega[i] \leftarrow \textsc{Count}(a[i], \texttt{SPF})$
end for
Initialize empty stack S, array $L^\star$, $R^\star$, *cap*
for $i \leftarrow 0$ to $n-1$ **do**
    while not $S.isempty()$ **and** $\Omega[S.\texttt{top}] < \Omega[i]$ **do**
        $S.\texttt{pop}()$
    end while
    $L^\star[i] \leftarrow \begin{cases} S.\texttt{top}, & S \neq \emptyset \\ 0, & \text{otherwise} \end{cases}$
    $S.\texttt{push}(i)$
end for
Clear stack S
for $i \leftarrow n-1$ down to 0 **do**
    while not $S.isempty()t$ **and** $\Omega[S.\texttt{top}] \leq \Omega[i]$ **do**
        $S.\texttt{pop}()$
    end while
    $R^\star[i] \leftarrow \begin{cases} S.\texttt{top}, & S \neq \emptyset \\ n{+}1, & \text{otherwise} \end{cases}$
    $S.\texttt{push}(i)$
end for
for $i \leftarrow 0$ to $n-1$ **do**
    $\texttt{cap}[i] \leftarrow (i - L^\star[i]) \times (R^\star[i] - i)$
end for
Build list $\mathcal{L} = \{(a[i], \texttt{cap}[i], i) \mid i = 1..n\}$
Sort $\mathcal{L}$ by decreasing $a[i]$; break ties by increasing i
$SUM \leftarrow 1$, $r \leftarrow k$
for each $(v, \texttt{cap}_i, i)$ in $\mathcal{L}$ **do**
    if $r = 0$ **then break**
    end if
    $t \leftarrow \min(\texttt{cap}_i, r)$
    $SUM \leftarrow SUM \times v^t$
    $r \leftarrow r - t$
end for
return SUM

Algorithm 5 SmallestPrimeFactorSieve(M)

```
for x ← 1 to M do
    SPF[x − 1] ← 0
end for
for p ← 2 to M do
    if SPF[p − 1] = 0 then
        SPF[p − 1] ← p
        if p · p ≤ M then
            for q ← p · p to M step p do
                if SPF[q − 1] = 0 then
                    SPF[q − 1] ← p
                end if
            end for
        end if
    end if
end for
return SPF
```

Algorithm 6 Count(x, SPF)

```
cnt ← 0
while x > 1 do
    p ← SPF[x − 1]
    cnt ← cnt + 1;    x ← x/p
end while
return cnt
```

(B) Proof of Correctness

Step 0: What the algorithm is solving. For each index i, let $\Omega(a_i)$ be the number of prime factors of a_i (with multiplicity). An interval $[L, R]$ "chooses" index i iff, inside $[L, R]$, $\Omega(a_i)$ is maximal and, among ties, i is the leftmost. The algorithm computes, for every i, exactly how many distinct intervals would choose i; call this number cap_i. Then it spends the k coins on the largest values a_i up to these capacities.

Step 1: Computing $\Omega(\cdot)$ correctly (SPF sieve & Count). Claim. After SmallestPrimeFactorSieve(M), for all $y \in \{2, \ldots, M\}$, SPF[$y-1$] equals the smallest prime factor of y; consequently, Count(x, SPF) returns $\Omega(x)$.

Proof (induction on y). Base $y = 2$: 2 is prime and the sieve sets SPF[1] = 2. Inductive step: assume true for $2, \ldots, y-1$. If y is unmarked, it is prime and the sieve sets SPF[$y-1$] = y. Otherwise y was first marked by a prime p when processing p, so $p \mid y$ and no smaller prime $p' < p$ could have marked y earlier; hence p is the smallest prime factor. Given correct SPF, Count divides by SPF[$x-1$] until $x = 1$, removing one prime factor per iteration, so it returns exactly $\Omega(x)$. □

Step 2: The left stack computes $L_i^\star$ (induction on i). Define

$$L_i^\star \;=\; \max\{\, j < i \;:\; \Omega(a_j) \geq \Omega(a_i) \,\} \quad \text{(or 0 if none)}.$$

scan $i = 0, \ldots, n-1$ and maintain a stack S with strictly decreasing Ω-values.

Invariant $\mathcal{I}(i)$. After finishing index i, S contains a subsequence of $\{0, \ldots, i\}$ in strictly decreasing Ω, and the value stored as $L_i^\star$ equals the nearest $j < i$ with $\Omega(a_j) \geq \Omega(a_i)$ (or 0 if none).

Proof. Base $i = 0$: the while-loop pops nothing, $L_0^\star = 0$ is correct, and $S = [0]$ satisfies strict decrease. Inductive step: assume $\mathcal{I}(i)$ holds. While $\Omega[S.\top] < \Omega[i{+}1]$ we pop; the first remaining top T (if any) then satisfies $\Omega(T) \geq \Omega(i{+}1)$ and is the nearest such index to the left; hence setting $L_{i+1}^\star \leftarrow T$ is correct. Pushing $i{+}1$ preserves strict decrease because any smaller Ω has been popped. □

Step 3: The right stack computes $R_i^\star$ (reverse induction). Define

$$R_i^\star \;=\; \min\{\, j > i \;:\; \Omega(a_j) > \Omega(a_i) \,\} \quad (\text{or } n{+}1 \text{ if none}).$$

Lines 15–21 scan $i = n{-}1, \ldots, 0$ and pop while $\Omega[S.\top] \leq \Omega[i]$.

Claim. After processing i, the value stored as $R_i^\star$ is the nearest index to the right with strictly larger Ω (or $n{+}1$ if none).

Proof (reverse induction on i). Base $i = n{-}1$: no right index exists, so $R_{n-1}^\star = n{+}1$. Inductive step: assume the claim holds for indices $> i$. Before pushing i, we pop all indices with $\Omega \leq \Omega[i]$; the first remaining top (if any) is the smallest $j > i$ with $\Omega(a_j) > \Omega(a_i)$ (any closer candidate with larger Ω would not be popped; any with $\leq$ is removed). Hence assigning $R_i^\star$ to that top (or $n{+}1$) is correct. □

Step 4: Capacities count exactly the valid intervals. By Steps 2–3, $[L, R]$ selects i iff $L_i^\star < L \leq i \leq R < R_i^\star$. There are $(i - L_i^\star)$ choices for L and $(R_i^\star - i)$ for R, so

$$\mathrm{cap}_i \;=\; (i - L_i^\star)(R_i^\star - i)$$

counts *exactly* the intervals that would choose i.

Step 5: Ordering of indices and coin allocation is optimal (induction on k). Let the indices be ordered so that $a_{i_1} \geq a_{i_2} \geq \cdots \geq a_{i_n}$ (ties by smaller index). For each t, let $\mathcal{I}_{i_t}$ be the set of intervals that select i_t; by Step 4, $|\mathcal{I}_{i_t}| = \mathrm{cap}_{i_t}$. Consider the following allocation rule:

Rule. For $t = 1, 2, \ldots$, use as many intervals from $\mathcal{I}_{i_t}$ as possible, up to $\min\{\mathrm{cap}_{i_t}, \text{remaining budget}\}$.

Claim. For any budget $k \geq 0$, this rule yields a maximum possible product.

Proof (induction on k). Base $k = 0$: product 1 is optimal. Inductive step: assume the claim holds for all smaller budgets. Let $t^\star$ be the least index with $\mathrm{cap}_{i_{t^\star}} > 0$. Consider any optimal plan $P^\star$ using k coins. If $P^\star$ uses a coin on some $\mathcal{I}_{i_s}$ with $s > t^\star$ while $\mathrm{cap}_{i_{t^\star}} > 0$, replace that single use by an unused interval from $\mathcal{I}_{i_{t^\star}}$; the plan remains feasible (capacities count disjoint intervals), and the product does not decrease because $a_{i_{t^\star}} \geq a_{i_s}$. Iterating this exchange yields an optimal plan that spends at least one coin on $\mathcal{I}_{i_{t^\star}}$. Fix one such use; the residual instance has budget $k{-}1$ with possibly reduced capacities but the same ordering. By the induction hypothesis, applying the same rule to the residual instance achieves an optimal product. Therefore the stated rule is optimal for budget k. □

Step 6: The returned value equals the intended score. The algorithm multiplies a_{i_t} exactly as many times as the rule prescribes (Step 5), never exceeding cap_{i_t} (Step 4) and never reusing an interval (capacities count disjoint intervals). Thus the product accumulated in lines 27–34 equals the maximum possible score after k coins.

Time Complexity Analysis. Let $n = |a|$ and $M = \max_i a[i]$.

- **SPF Sieve :** $O(M \log \log M)$.
- **Prime factor counts :** $O(n \log M)$.
- **Stacks for $L^\star, R^\star$:** $O(n)$.
- **Capacities and list build:** $O(n)$.
- **Sorting:** $O(n \log n)$.
- **Coin distribution:** $O(n)$.

Overall:

$$T(n, M) = O(M \log \log M + n \log M + n \log n) = O\big(n \log n\big)$$

Question 3: Wealth Accumulate

We have a binary tree, n nodes, initial balances $W^{(0)} \in \mathbb{R}^n_{\geq 0}$. Each year $t = 1, 2, \ldots$:

- Every node wealth quadruples.
- Then each node donates *half of its (post-growth) wealth* to exactly one child: left on odd years, right on even years, done simultaneously.

Let $W^{(t)}$ be the wealth vector *after* year t completes.

Let A_L be the $n \times n$ matrix that maps a parent's wealth to its left child (i.e., $(A_L)_{child,parent} = 1$ if *child* is the left-child of *parent*, else 0). Define A_R analogously for right edges. Then one year with left-donation is:

$$W^{(t)} \;=\; (2I \;+\; 2A_L)\, W^{(t-1)} \quad (\text{odd } t),$$

and one year with right-donation is:

$$W^{(t)} \;=\; (2I \;+\; 2A_R)\, W^{(t-1)} \quad (\text{even } t).$$

Hence, for k years (starting with $t = 1$ odd/left),

$$W^{(k)} \;=\; \Big(\prod_{t=1}^{k} \big(2I + 2A_{\sigma(t)}\big)\Big)\, W^{(0)}, \quad \text{where } \sigma(t) = \begin{cases} L, & t \text{ odd} \\ R, & t \text{ even.} \end{cases}$$

(a) Algorithm

The schedule alternates $L, R, L, R, \ldots$ so two consecutive years form a fixed operator

$$B \;=\; (2I + 2A_R)(2I + 2A_L) \;=\; 4I + 4A_L + 4A_R + 4A_R A_L.$$

Thus

$$W^{(k)} = \begin{cases} B^{\lfloor k/2 \rfloor} \big(2I + 2A_L\big)\, W^{(0)}, & k \text{ odd}, \\ B^{k/2}\, W^{(0)}, & k \text{ even}. \end{cases}$$

Algorithm 7 MatrixMultiply(A, B, n)

```
Initialize C[1..n][1..n] ← 0
for i ← 0 to n − 1 do
    for j ← 0 to n − 1 do
        for k ← 0 to n − 1 do
            C[i][j] ← A[i][k] · B[k][j]
        end for
    end for
end for
return C
```

Algorithm 8 MatVecMultiply(A, x, n)

```
Initialize y[1..n] ← 0
for i ← 0 to n − 1 do
    s ← 0
    for j ← 0 to n − 1 do
        s ← s + A[i][j] · x[j]
    end for
    y[i] ← s
end for
return y
```

Algorithm 9 Identity(n)

```
Initialize I[1..n][1..n] ← 0
for i ← 0 to n − 1 do
    I[i][i] ← 1
end for
return I
```

Algorithm 10 MatrixPower(A, e, n)

```
R ← IDENTITY(n)
P ← A
while e > 0 do
    if e mod 2 = 1 then
        R ← MATRIXMULTIPLY(R, P, n)
    end if
    e ← ⌊e/2⌋
    if e > 0 then
        P ← MATRIXMULTIPLY(P, P, n)
    end if
end while
return R
```

Algorithm 11 BSTToWealth_Simple(*root*)

if *root* = `null` **then**
  return 0, empty arrays
end if
Initialize empty queue Q
$n \leftarrow 0$
Enqueue *root* into Q
while Q not empty **do**
  $u \leftarrow Q$.`pop_front`()
  $n \leftarrow n + 1$; u.`id` $\leftarrow n$
  $W^{(0)}[n] \leftarrow u$.`wealth`
  if u.`left` $\neq$ `null` **then**
    Q.`push_back`(u.`left`)
  end if
  if u.`right` $\neq$ `null` **then**
    Q.`push_back`(u.`right`)
  end if
end while
Initialize `leftIdx`$[0..n-1] \leftarrow 0$, `rightIdx`$[0..n-1] \leftarrow 0$
Enqueue *root* into Q
while Q not empty **do**
  $u \leftarrow Q$.`pop_front`(); $i \leftarrow u$.`id`
  if u.`left` $\neq$ `null` **then**
    `leftIdx`$[i] \leftarrow u$.`left.id`; Q.`push_back`(u.`left`)
  end if
  if u.`right` $\neq$ `null` **then**
    `rightIdx`$[i] \leftarrow u$.`right.id`; Q.`push_back`(u.`right`)
  end if
end while
return n, `leftIdx`, `rightIdx`, $W^{(0)}$

Algorithm 12 BuildAL_AR(n, `leftIdx`, `rightIdx`)

Initialize $A_L \leftarrow 0_{n\times n}$, $A_R \leftarrow 0_{n\times n}$
for $p \leftarrow 0$ to $n-1$ **do**
  if `leftIdx`$[p] \neq 0$ **then**
    $\ell \leftarrow$ `leftIdx`$[p]$; $A_L[\ell][p] \leftarrow 1$
  end if
  if `rightIdx`$[p] \neq 0$ **then**
    $r \leftarrow$ `rightIdx`$[p]$; $A_R[r][p] \leftarrow 1$
  end if
end for
return A_L, A_R

Algorithm 13 FinalWealth($root, k$)

$(n, \texttt{leftIdx}, \texttt{rightIdx}, W^{(0)}) \leftarrow$ BSTWEALTH($root$)
$(A_L, A_R) \leftarrow$ BUILDAL_AR($n, \texttt{leftIdx}, \texttt{rightIdx}$)
$I \leftarrow$ IDENTITY(n)
$Lop \leftarrow 2I + 2A_L$
$Rop \leftarrow 2I + 2A_R$
$B \leftarrow$ MATRIXMULTIPLY(Rop, Lop, n)
if $k \bmod 2 = 0$ **then**
    $Bpow \leftarrow$ MATRIXPOWER($B,\ k/2,\ n$)
    $W^{(k)} \leftarrow$ MATVECMULTIPLY($Bpow,\ W^{(0)},\ n$)
else
    $Bpow \leftarrow$ MATRIXPOWER($B,\ \lfloor k/2 \rfloor,\ n$)
    $tmp \leftarrow$ MATVECMULTIPLY($Lop,\ W^{(0)},\ n$)
    $W^{(k)} \leftarrow$ MATVECMULTIPLY($Bpow,\ tmp,\ n$)
end if
return $W^{(k)}$

(b)Complexity

Time Complexity. Let all matrices be $n \times n$ and vectors be length n.

- **MatrixMultiply**: Three nested loops over $i, j, k \Rightarrow$ time $T_7 = O\big(n^3\big)$.
- **MatVecMultiply**: Two nested loops over $i, j \Rightarrow$ time $T_8 = O\big(n^2\big)$.
- **Identity**: One double loop touching diagonal once $\Rightarrow$ time $T_9 = O\big(n^2\big)$.
- **MatrixPower**: Binary exponentiation does $O(\log k)$ matrix multiplications and squarings. Each multiply is $O\big(n^3\big)$. Thus
$$T = O(n^3 \log k)$$
- **BSTToWealth**: Two BFS passes over n nodes/edges $\Rightarrow$ time $T = O\big(n\big)$ for the queue and arrays ($W^{(0)}$, left/right indices).
- **BuildAL_AR**: Single pass over $p = 0..n-1$ with $O(1)$ work per node; writing into dense $A_L, A_R \Rightarrow$ time $T = O\big(n\big)$.
- **FinalWealth**:
  - Build I, $Lop = 2I + 2A_L$, $Rop = 2I + 2A_R$: $O(n^2)$.
  - Compute $B = Rop \cdot Lop$: one matrix multiply $\Rightarrow O(n^3)$.
  - Exponentiate: $B^{\lfloor k/2 \rfloor}$ via Alg. 10 $\Rightarrow O(n^3 \log k)$.
  - One or two matrix–vector multiplies $\Rightarrow O(n^2)$.

  Therefore
$$T = O\big(n^3 \log k\big) \quad \text{(dominant)},$$

Summary. The overall cost is dominated by matrix exponentiation:

$$\boxed{T = O\big(n^3 \log k\big)}.$$

Question 4: The King's Punishment

Let the knights' heights be $h[1..n]$, all distinct. For knight i, the number of punishments equals the number of $j < i$ with $h[j] > h[i]$.

(a) Brute Force

Algorithm:

1. Let the knights' heights be stored in an array $H[1 \dots n]$, where $H[i]$ is the height of the i-th knight in line.
2. Initialize an array $P[1 \dots n]$ with all zeros, where $P[i]$ will store the number of punishments for knight i.
3. For each knight i from 1 to n:
   (a) Compare $H[i]$ with every previous knight $H[j]$ for $j = 1 \dots i-1$.
   (b) If $H[j] > H[i]$, then increment $P[i]$ by 1.
4. Return the array P.

Algorithm 14 Brute Force Punishment Calculation

```
Initialize array P[1..n] ← 0
for i ← 1 to n do
    for j ← 1 to i − 1 do
        if H[j] > H[i] then
            P[i] ← P[i] + 1
        end if
    end for
end for
return P
```

Time Complexity:

- For each knight i, the algorithm scans all $i-1$ knights before him.
- Total comparisons:
$$\sum_{i=1}^{n}(i-1) = \frac{n(n-1)}{2} = O(n^2).$$
- Thus, the brute force algorithm runs in $O(n^2)$ **time**.

Time: $O(n^2)$.

(b) Data Structure and Pseudocode: BST with Right-Subtree Counts

Data Structure. We maintain a Binary Search Tree (BST) keyed by height (all heights distinct). Each node stores:

$$\text{Node} = \langle \text{key} : \text{height},\ \text{left},\ \text{right},\ \text{rightSize},\ \text{punish} \rangle$$

- `rightSize`: number of nodes in the right subtree of this node.
- `punish`: the number of punishments for the knight (fixed at insertion time).

Key Observation. When inserting a height h, the number of taller prior knights equals the number of nodes in the BST with keys $> h$. During insertion, whenever we visit a node with key $> h$ (i.e., a taller knight), we add

$$1 + \text{rightSize(node)}$$

to the running total because the current node (1) and *all* nodes in its right subtree (rightSize) are also taller than h.

Conversely, when we move right (i.e., current node's key $< h$), we will be inserting into its right subtree, so we increment that node's `rightSize` by 1 along the path to keep subtree counts correct.

Algorithm 15 BST Node Definition

```
type Node:
    key: integer
    left, right: Node pointers
    rightSize: integer
    punish: integer
```

Algorithm 16 InsertAndComputePunish(root, h)

```
1: if root = null then
2:     u ← new Node
3:     u.key ← h; u.left ← null; u.right ← null
4:     u.rightSize ← 0; u.punish ← 0
5:     return (u, 0)
6: end if
7: curr ← root;  parent ← null;  pun ← 0
8: while curr ≠ null do
9:     parent ← curr
10:    if h < curr.key then
11:        pun ← pun + 1 + curr.rightSize
12:        curr ← curr.left
13:    else
14:        curr.rightSize ← curr.rightSize + 1
15:        curr ← curr.right
16:    end if
17: end while
18: u ← new Node
19: u.key ← h; u.left ← null; u.right ← null
20: u.rightSize ← 0; u.punish ← pun
21: if h < parent.key then
22:    parent.left ← u
23: else
24:    parent.right ← u
25: end if
26: return (root, pun)
```

Algorithm 17 BuildAllPunishments($H[1 \dots n]$)

```
1: root ← null
2: for i ← 1 to n do
3:     (root, P[i]) ← InsertAndComputePunish(root, H[i])
4: end for
5: return P
```

(c) Ensuring Operations Remain Efficient

Yes. If we keep the knights in a normal Binary Search Tree, the tree can become *unbalanced* (for example, if the knights arrive in sorted order). In that case, the tree behaves like a linked list, and both insertion and query would take $O(n)$ time instead of $O(\log n)$.

To avoid this, we need a balanced structure:

- Use a **self-balancing BST**. Balancing ensures the height of the tree stays $O(\log n)$.

Conclusion: We must make sure the data structure is balanced (or use a balance-free one) so that both updates and queries remain $O(\log n)$ as the number of knights grows.

(d) Time Complexity Analysis

In our data structure, each knight is inserted one by one. For every insertion we need to:

- **Update:** When we insert a knight, we update the data structure (Balanced BST). This takes $O(\log n)$.
- **Query:** To compute the punishments, we query how many taller knights already exist. This also takes $O(\log n)$.

Thus, each iteration performs two $O(\log n)$ operations. Over all n knights, the total running time is:

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

End of Solutions.