# Painter's Partition Problem

GROUP MEMBERS:

U19CS066 Mahitha Gurrala

U19CS068 Guntuboina Venkata Sankirtana

U19CS076 Krithikha Balamurugan

# PROBLEM STATEMENT

- There are N boards of Length {L1,L2,...Ln} which are to be painted.
- There are K painters available and each painter takes 1 unit of time to paint 1 unit of board.
- Find the minimum Time Taken to get this job done. There is a Constraint That Painters will only paint contiguous sections of boards(a configuration where painter 1 paints board 1 and 3 but not 2 is invalid).

# INPUT

First line of input consist of an integer n denoting number of boards to be painted that is, number of elements in array A.
Second line of input consist of an integer k denoting number of painters.
Next line of input consist of array A which denotes n partitions of length Ai to be painted.

# OUTPUT

Minimum Time to get job done considering one unit board takes one unit time under the given constraints.

# Understanding the problem

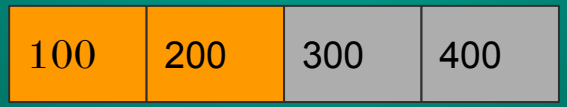
Given an array A of non-negative integers and a positive integer k, we have to divide A into k of fewer partitions such that the maximum sum of the elements in a partition, overall partitions is minimized. For example let's consider 4 boards and 2 painters.

**One partition:**

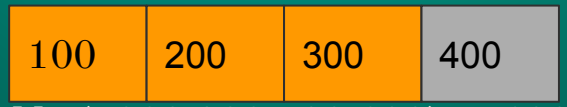

Max Time = 1000

**Two partitions:**


Max Time = 900




Max Time = 700




Max Time = 600

Min(1000,900,700,600) = 600

# SAMPLE TEST CASE

➔ **Input :** k = 2, A = {5, 5, 5, 5}
**Output :** 10.

Here we can divide the boards into 2 equal sized partitions, so each painter gets 10 units of board and the total time taken is 10.

➔ **Input :** k = 1, A = {2, 7, 9, 1}
**Output :** 19

Here we cannot divide the boards into partitions, so the one painter gets all units of board and the total time taken is 19.

➔ **Input :** k = 2, A = {10, 20, 30, 40}
**Output :** 60.

Here we can divide first 3 boards for one painter and the last board for second painter.

# EXPLANATION

➔ **Input :** k = 2, A = {5, 5, 5, 5}
**Output :** 10

Here we can divide the boards into 2 equal sized partitions, so each painter gets 10 units of board and the total time taken is 10. ->Minimum time

Showing some ways to partition the work->

Painter 1 - {A1,A2}- {5,5} = 10 Time to complete work =10
Painter 2- {A1,A2}- {5,5} = 10

Or

Painter 1 - {A1,A2,A3}- {5,5,5} = 15 Time to complete work=15
Painter 2- {A1}- {5} = 5 ->Max of (5,15)

Or

Painter 1 - {A1,A2,A3,A4}- {5,5,5,5} =20 Time to complete work=20
Painter 2- {} = 0 ->Max of (5,20)

# Understanding the Approach

**Goal:** To divide the paintings among painters so that work Is done in minimum time possible

If we have equal lengths. Then simply We can divide equally

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| 10 | 10 | 10 | 10 | 10 | 10 | 10 | 10 | 10 |

But what if the boards are of different lengths? But this way it's unfair for the painters

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| 10 | 20 | 30 | 40 | 50 | 60 | 70 | 80 | 90 |

The above allocation is clearly not appropriate, as it is extremely unfair to the third painter.

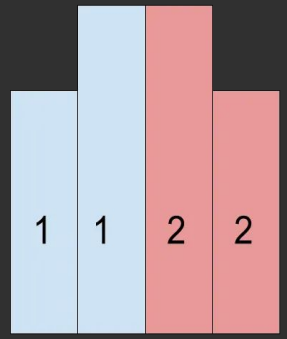
| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| 10 | 20 | 30 | 40 | 50 | 60 | 70 | 80 | 90 |

Therefore the above allocation is the fairest possible way to divide the boards among the three painters .

# APPROACHES

Solving the problem in the following ways and finding the optimal solution which satisfies the Minimum Time and Space complexity constraints.

- Recursive
- Bottom Up
- Binary search




Painter 1 paints fence 1 and 2.
Painter 2 paints fence 3 and 4.
Any other arrangement will increase the total time.

# Approach 1: Recursive

**Optimal Substructure**

A problem has an **optimal substructure** if its optimal solution can be constructed from the optimal solutions of its subproblems.

In this we can see we are dividing the array into different partitions and we were calling the partition on the smaller subarray similarly solving the smaller subarray and using the already summed subarray value we can solve the entire array value. Therefore dividing a problem into smaller subproblems to get the solution for the larger problem.

we already have k-1 partitions in place (using k-2 dividers), we now have to put the k-1 th divider to get k partitions. We can put the k-1 th divider between the i th and i+1 th element where i = 1 to n.

# Approach 1: Recursive

Implementing Recursive solution using optimal substructure property by following **Recurrence Relation.**

$$T(n,k) = min\left\{max_{i=1}^{n}\left\{T(i,k-1), \sum_{j=i+1}^{n} A_j\right\}\right\}$$

- Calculate maximum subarray sum for all possible combinations
- Take minimum value of these maximum values found. This will give the best possible painting time.

# Approach 1: Recursive

**Base Case/ Termination Condition:**

Base cases are when there is a single painter or single painting.

If T(1,k) Which means there is only 1 painting, Then return A[0] because we can't share single painting among different painters.

$$T(1,k) = A[0]$$

Else If T(n,1) Which means there is only 1 painter,Then return the sum of all the time taken to paint all the paintings because only a single painter will be working on all paintings.

$$T(n,1) = \sum_{i=1}^{n} A_i$$

# Approach 1: Recursive

**Implementation of the Equation of the Recurrence Relation:**

- Let's set the Best value as maximum Integer.
- The maximum sum can be found using a simple **helper function** to calculate sum of elements between two indices in the array.
- Find minimum of all possible maximum k-1 partitions to the left of arr[i] . With i elements, put k-1 th divider between arr[i-1] & arr[i] to get k-th partition .

# Dry Run

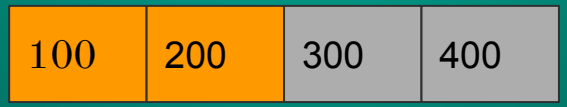
For example let's consider 4 boards and 2 painters. [T(4,2) part]

**One partition:** It will be the base condition and 1 painter will be working on all paintings


Max Time = 1000 Min = 1000

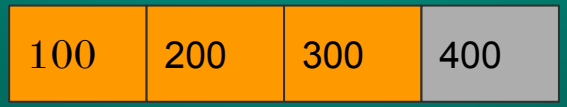
**Two partitions:** Now for 2 partitions let's place a partition in different locations in array .


Max Time = 900 Min = 900




Max Time = 700 Min = 700

Min(1000,900,700,600) = 600




Max Time = 600 Min = 600

After finding Max sum for all the configurations then find the minimum among all the max values.

# Pseudocode

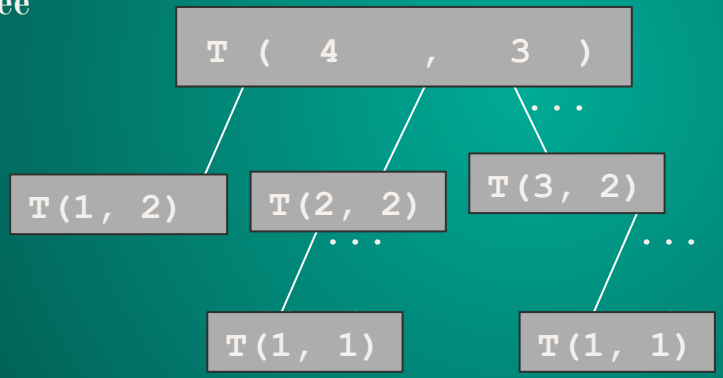
```
  int partition(int[] arr, int n, int k)
  {
    // base cases
1.  if  k =1 // one partition
2.       return sum(arr, 0, n - 1);

3.  if n =1//one board
4.        return arr[0];

5.  set best = INT_MAX;

6.  for i=1 to i=n
7.      best = min(best, max(partition(arr, i, k - 1),sum(arr, i, n - 1)));

8.  return best;
   }
}
```

# Time and Space complexity Analysis

**Time complexity** is **O(N^K)** which is exponential and **Space complexity** is **O(n)** (Due to call stack)

K ---- number of painters  N ---- number of boards

**The Recursive Tree**




As we can observe that many overlapping subproblems like T(1,1) are being solved again and again.

Therefore To reduce the time complexity we can solve it using Dynamic Programming Approach.

# Optimization

Though here we consider to divide A into k or fewer partitions, we can observe that the optimal case always occurs when we divide A into exactly k partitions. So we can optimize the code and modify the other implementations accordingly.

```
for (int i = k-1; i <= n; i++)
    best = min(best, max( partition(arr, i, k-1),
                          sum(arr, i, n-1)));
```

# Approach 2: Bottom – up method

**What is Tabulation or bottom up approach?**

Describe a state for a Dynamic Programming problem to be dp[x] with dp[0] as base state and dp[n] as our destination state. So, we need to find the value of destination state that is dp[n].

If we start our transition from our base state i.e dp[0] and follow our state transition relation to reach our destination state dp[n], we call it Bottom Up approach as it is quite clear that we started our transition from the bottom base state and reached the top most desired state.

# Pseudo Code

```
1.int findMax(A[], n, k){
2.      dp[k + 1][n + 1] = { 0 };
3.      for i = 1 to i= n; i++{
4.          dp[1][i] = sum(A, 0, i - 1);
5.      }
6.      for i = 1 to i= k; i++{
7.          dp[i][1] = A[0];
8.      }
9.      for i = 2 to i= k; i++{ // 2 to n boards
10.         for j=2 to j=n; j++ {
11.             best = INT_MAX;
12.             for p=1 to p=j; p++{
13.                 best = min(best, max(dp[i - 1][p], sum(A, p, j - 1)));
14.             }
15.             dp[i][j] = best;
16.         }
17.     }
18.     return dp[k][n];
19.}
```

# Pseudo Code

In the pseudo code, the sum function is defined as:

```
1. int sum(arr[], from, to){
2.     total = 0;
3.     for i = from to i= to; i++{
4.         total += arr[i];
5.     }
5.     return total;
6. }
```

# DRY RUN

Let's consider 6 paintings of lengths {10,40,20,30,40,50} and 3 painters

| | 0 | 1 | 2 | 3 | 4 | 5 | 6 |
|---|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 1 | 0 | 10 | 50 | 70 | 100 | 140 | 190 |
| 2 | 0 | 10 | 40 | 50 | 50 | 70 | 100 |
| 3 | 0 | 10 | 40 | 40 | 50 | 50 | 70 |

A

nswer = DP[3][6] = 70

# Dynamic Programming Approach

Sum()

Time complexity = $C_1 + C_2(n) + C_3(n-1) + C_4$

$= O(n)$

FindMax()

Time complexity = $C_1 + C_2(n) + C_3(K) + C_4(k)^2 [C_5(n)^3 [C_6 + C_7(n) [C_8(n-1)] + C_9]] + C_{10}$

$= C_1 + C_2 n + C_3 K + K * C_4 [C_5 n + C_6 n^2 + C_7 n^3]$

$= C_1 + C_2 n + C_3 K + C_a(Kn) + C_b K n^2 + C_c K n^3$

$= O(Kn^3)$

# Time Complexity

The time complexity of the above program is $O(k*N^3)$. It can be easily brought down to $O(k*N^2)$ by precomputing the cumulative sums in an array thus avoiding repeated calls to the sum function -> shown in next slide .

# Time Complexity Optimization

```
int sum[n+1] = {0};
 // sum from 1 to i elements of arr
 for (int i = 1; i <= n; i++)
   sum[i] = sum[i-1] + arr[i-1];
 for (int i = 1; i <= n; i++)
   dp[1][i] = sum[i];
and using it to calculate the result as:
best = min(best, max(dp[i-1][p], sum[j] - sum[p]));
```

# Space Complexity

Space Complexity of Bottom Up dynamic programming approach is O(n*k).

Therefore, we have a polynomial space complexity solution for this approach

# Approach 3: Using Binary Search

Here we are going to code the most optimized algorithm using binary search.

We know that the invariant of binary search include two points

1. The loop will decrease the range of search each time till termination reaches.
2. The target value to be found will always be in searching range. Here our target refers to the maximum sum of contiguous section of array in the optimal allocation of boards.

So since we know our target value to be found from array, we can fix both the possible limits of range for our target and narrow it down to get the desired output.

# Understanding the Approach

Going by the understanding we have for the problem statement we conclude this:

1. If we have more number of painters than the number of boards to be painted **(k>=n)**, then the minimum time for all painters to paint all the boards will be to paint the maximum length board. **-> low**
2. If we have only one painter **(k=1)**, then that person has to paint all the boards, so minimum time it will take will be the sum of the lengths of the boards **->high**

So using this approach we get the upper limit (high) and lower limit (low) of minimum time and do a "Binary Search" over the board units to be painted. The search space will be the range of [low, high] as explained above.

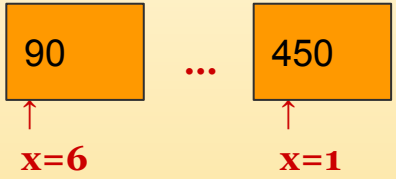
# Sample input & Explanation & Dry run

Let us study using the example below to understand how this works:
Input - 9
3
10 20 30 40 50 60 70 80 90
Explanation
We see array A = { 10, 20, 30, 40, 50, 60, 70, 80, 90} and k = 3.

The highest possible result must be the sum of A, 450. (ie,assigning all boards to one painter).

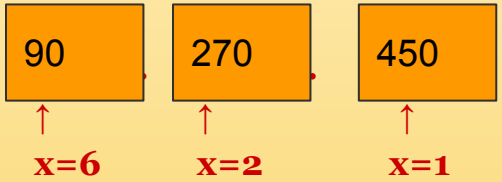
The lowest possible result must be the largest element in A, 90. This requires a total of six painters — The first painter paints {10, 20, 30}, second painter paints {40, 50} while the rest of them paints one board each).

Below is a simple conceptual illustration of how the search space looks like, with its corresponding x value (the required number of painters) pointing to minimum time.
We want to find the minimum time under the constraint of x = k.



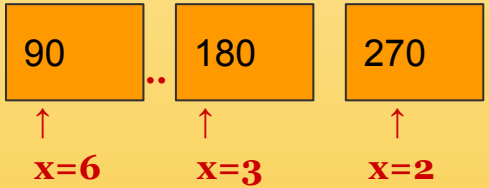

**Note that x decreases while minimum time increases.**

We choose the middle element, 270, and find its corresponding x, which is 2.



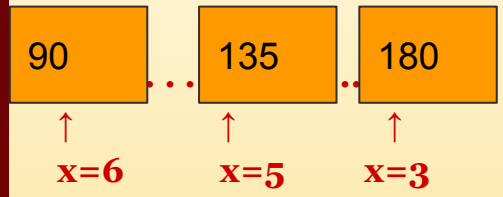

**[2<3 so high=270]**

Since **[2<3 so high=270]** we discard the upper half and continue searching in the lower half.




**[3 is still not greater than 3 so high=180]**

The middle element now is 180, discard the upper half again and continue searching in the lower half.




**[5>3 so low=mid+1 so low=136]**

The lower half including value 135 will be discarded (to maintain the variant).So let's continue searching in upper half [136,180].

**After multiple successions of halving the search space,**
**the final answer = 170,**
**its corresponding x = 3.**

This is also the minimum of time while maintaining the requirement x = k.

From this we can find the target value when x=k and use a helper function to find x, the minimum number of painters required when the maximum length of section a painter can paint is given.

# Pseudo Code-partition_range

```
1.   int partition_range(int arr[], int n, int k) {                c1
2.     low = getMax(arr,n); // lower limit of min time            c2
3.     high = getSum(arr,n); // upper limit of min time           c3
4.     while (low < high) {                                       c4
5.         mid = (low + high) / 2;                                c5
6.       // get number of painters with max paint limit as mid
7.         req = painter_count(arr,n, mid);                       c6
8.         if (req <= k)                                          c7
9.             high = mid ;                                       c8
10.        Else                                                   c9
11.            low = mid+1;                                       c10
12.    }
13.    return low;                                                c11
14.  }
```

# Pseudo Code-painter_count function

```
1. int painter_count(int arr[], int n, int l) {
2.  total = 0, num = 1;    //num stores number of painters  c1
3.  for i=0 to i=n-1,increment 1                            c2
    {
4.      total += arr[i];                                    c3
5.      if (total > l)                                      c4
        {
6.          total = arr[i];                                 c5
7.          num+=1;                                         c6
        }
    }
8.  return num;                                            c7
9.}
```

Binary_Search approach

Painter_count()

Time Complexity = $C_1 + C_2(n) + C_3(n-1) + C_4(n-1) + C_5(n-1) + C_6(n-1) + C_7$

$= A(n) + B$

$= O(n)$

Partition_range()

Time Complexity ⇒

The size of search range is from low to high

According to binary search algorithm, (invariant) painter_counter() is called log(high) times upper bound

Time Complexity = $C_1 + C_2(n) + C_3(n) + [\log(high) * [(C_6 * O(n)) + C_7 + C_8$ (as high = sum(arr[])) $+ C_9 + C_{10}] + C_{11}$

$= \log(sum(arr[])) \times O(n)$

$= O(n \times \log(sum(arr[])))$

# Time and Space Complexity

The binary search algorithm breaks the list down in half on every iteration.The range of array goes till high which is sum of all array elements.

The painter_count function which finds the the minimum number of painters required when the maximum length of section a painter can paint is given has time complexity of O(n).

**The time complexity of the above approach is  O(n*log(sum(arr[])).**

**Space complexity is O(1)**

# CONCLUSION

We initially started off with the recursive approach for the problem but it proved to be not so efficient as it had an exponential time complexity of $O(N^k)$.

Next we tried to have a dynamic programming approach bringing time complexity to $O(k*N^3)$ but it increased the space complexity to O(k*N) extra space.After trying our modifications in dynamic code we could bring it to $O(k*N^2)$.

So we want to reduce the extra space and possibly the time complexity. So finally, we tried the binary search approach where we calculate high and low ranges of possible minimum time .This proved to be the most optimum approach with lesser space and time complexity. The time complexity was thus brought down to

**O(n*log(sum(arr[])).**

# CONCLUSION

We have tried different approaches and were successful in finding the best solution for the problem statement. We have learnt a lot from the practical difficulties we faced during the project and we have overcome those all today by perseverance. We have all learnt the different approaches to tackle this problem.

# Applications and Other examples

The same approach can be used in many other problems like : ALLOCATE MINIMUM PAGES, COLLECT COINS FROM HORIZONTAL STACK  IN MINIMUM STEP, ,Search a pattern using the built Suffix Array .The approach of Binary search is used in 3D games and apps.Every sorted collection in every language library uses this approach.Even machine learning algorithms use it to find values which correspond to another.

Thank you