

PAINTER'S PARTITION PROBLEM

Team:

Mahitha Gurrala	U19CS066
Guntuboina Venkata Sankirtana	U19CS068
Krithikha Balamurugan	U19CS076

ACKNOWLEDGMENT

In pursuit of this academic endeavor, we feel that we have been fortunate; inspiration, guidance, direction, cooperation, love and care, all came our way in abundance and it seems almost an impossible task for us to acknowledge the same in adequate terms. Yes, we shall be failing in our duty if we do not record the profound sense of indebtedness and heartfelt gratitude to our teacher, Dr. Keyur Parmar who guided and inspired us in pursuance of this work. His association will remain a beacon of light to us throughout our life. We owe a depth of gratitude to the college, for all the facilities provided. We thank for the support, encouragement and good wishes of our parents and family members, without which we would not have been able to complete our report.

-

Krithikha Balamurugan *U19CS076*
Mahitha Gurralla *U19CS066*
Guntuboina Venkata Sankirtana *U19CS068*

INDEX

Topic	Page Number
Problem Statement	4
Approach 1	6
Approach 2	14
Approach 3	19
Conclusion	28

PROBLEM STATEMENT

There are N boards of Length $\{L_1, L_2, \dots, L_n\}$ which are to be painted. There are K painters available and each painter takes 1 unit of time to paint 1 unit of board.

The problem is to 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 consists of an integer n denoting number of elements in array A . This denotes the number of boards of varying length to be painted

Second line of input consists of an integer k denoting the number of painters.

Next line of input consists of array A which denotes n boards of length A_i to be painted.

Output

Minimum Time to get a job done under given constraints.

Sample Test Case

→ **Input :** $k = 2, A = \{5, 5, 5, 5\}$

Output : 20.

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 the first 3 boards for one painter and the last board for the second painter.

Explanation

→ **Input :** $k = 2, A = \{5, 5, 5, 5\}$

Output : 20

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 all 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 - $\{A1\}$ - $\{5\} = 5$

Time to complete work = 15

Painter 2 - $\{A1, A2, A3\}$ - $\{5, 5, 5\} = 15$

-> 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)

Or

Painter 2 - $\{\}$ = 0

Time to complete work = 20

Painter 1 - $\{A1, A2, A3, A4\}$ - $\{5, 5, 5, 5\} = 20$

-> Max of (5, 20)

Understanding The problem Deeply

All painters work at the same speed and in 1 unit of time a painter can paint 1 unit of a painting and a painter can only paint the adjacent paintings.

For example let's take 9 boards and 3 painters and let's check for two cases

CASE 1: If we have equal lengths. We can divide equally

10 10 10 | 10 10 10 | 10 10 10

CASE 2: But what if the boards are of different lengths?

10 20 30 | 40 50 60 | 70 80 90

But this way it's not dividing the work in a fair manner and the 3rd painter is getting extra time .

If the Boards are partitioned in the following manner

10 20 30 40 50 | 60 70 | 80 90

In this way, the largest board to paint is 170 units whilst the smallest board to paint is 130 units. Compared to the previous allocation, this allocation allows painting to finish faster because the maximum over the partitions has decreased from 240 to 170 units.

APPROACHES

1. **Recursive**
2. **Bottom Up**
3. **Binary search**

Recursive Approach

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:

Take boards of lengths {100,200,300,400} and let 2 painter's finish the job

- One partition: so time is 1000.
- Two partitions:
 - a. (100) & (200, 300, 400), max time is 900
 - b. (100,200) & (300,400), max time is 700
 - c. (100,200,300) & (400), max time is 600

so this means the minimum time: (1000, 900, 700, 600) is 600.

Solution is To :

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

Optimal Substructure:

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

Finding for optimal Substructure for the given problem:

- Considering the solution given above we can assume that we already have K-1 partitions in place (the left Partition) and we are trying to put the last separator (the (K-1)th separator).
- The possible locations to put the last separator would be between (i-1)th and i th element where i = 1 to N.
- Once we have the last separator in place, we have completed all k partitions.

The total cost of this arrangement can be calculated as the **maximum** of the following:

1. The cost of the last partition $\text{sum}(A_i..A_n)$, where the k-1 th divider is before element i.
2. The cost of the largest partition formed to the left of i

The 1) part can be found using a simple **helper function** to calculate the sum of elements between two indices in the array.

To find the 2) part we can place the remaining (K-2) separators such that the left partition is divided as fairly as possible, which leads to another valid sub-problem itself.

Thus we can write the optimal substructure property as the following recurrence relation:

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

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 a 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.

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

Implementation of the Equation of the Recurrence Relation:

- Let's set a Best value as maximum Integer.
- find minimum of all possible maximum sum of k-1 partitions to the left of arr[i].
- with i elements, put a k-1 th divider between arr[i-1] & arr[i] to get the k-th partition.

Code:

```
#include<bits/stdc++.h>
#include <time.h>
using namespace std;
clock_t begin, end;
double time_;
int sum(int arr[], int h, int l)
{
    int tot = 0;
    for (int i = h; i <= l; i++)
        tot += arr[i];
    return tot;
}

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

    int best = INT_MAX;

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



```
    }
    return best;
}

int main()
{
    int n, k, i;
    int *arr;
    cout << "Enter the number of boards of varying length to be painted: ";
    cin >> n;
    cout << "Enter the number of painters available: ";
    cin >> k;
    cout << "Enter the lengths of " << n << " partitions : ";
    arr = (int*) malloc (n * (sizeof(int)));
    for (i = 0; i < n; i++) {
        cin >> arr[i];
    }
    cout << "-----";
    begin = clock();
    cout << endl << "Minimum time to paint all boards: " << partition(arr, n, k)
    << endl;
    end = clock();
    time_ = ((double) (end - begin)) / CLOCKS_PER_SEC;
    cout << "\nTime taken : " << time_;

    return 0;
}
```

OUTPUT:

```
D:\svn\sem4\daa\painter_recursive.exe
Enter the number of boards of varying length to be painted: 9
Enter the number of painters available: 3
Enter the lengths of 9 partitions : 1 2 3 4 5 6 7 8 9
-----
Minimum time to paint all boards: 17

Time taken :0
-----
Process exited after 6.297 seconds with return value 0
Press any key to continue . . .
```

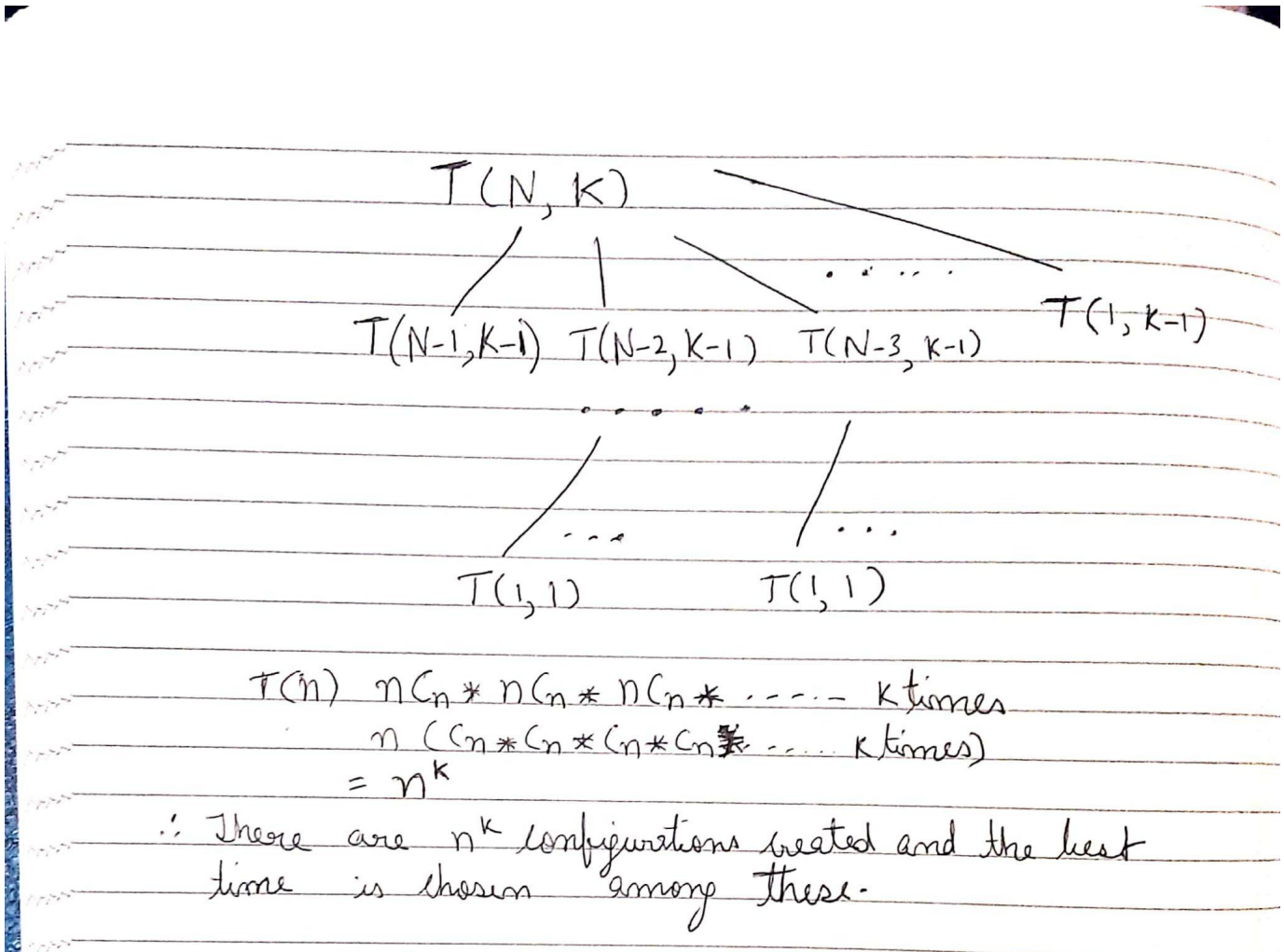
```
D:\svn\sem4\daa\painter_recursive.exe
Enter the number of boards of varying length to be painted: 20
Enter the number of painters available: 12
Enter the lengths of 20 partitions : 1 2 3 4 5 6 7 8 9 10 11 12 32 24 43 35 54 23 43 44
-----
Minimum time to paint all boards: 54

Time taken :1.457
-----
Process exited after 38.95 seconds with return value 0
Press any key to continue . . .
```

Time Complexity

Time complexity is $O(N^K)$ which is exponential because there are N^K different configurations possible and **Space complexity** is $O(n)$ (Due to call stack).

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

Time Complexity AnalysisAnother Approach

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. We can reduce the number of loops by implementing this approach.

C++ Code

```

#include<bits/stdc++.h>
using namespace std;

```

```
// function to calculate sum between two indices
int sum(int arr[], int from, int to)
{
    int total = 0;
    for (int i = from; i <= to; i++)
        total += arr[i];
    return total;
}

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

    int best = INT_MAX;
    //optimal case always occurs when we divide A into exactly k partitions
    for (int i = k-1; i <= n; i++){
        best = min(best, max(partition(arr, i, k - 1), sum(arr, i, n - 1)));
    }
    return best;
}


int main(){
    int n, k, i;
    int *arr;
    cout << "Enter the number of boards of varying length to be painted: ";
    cin>>n;
    cout<<"Enter the number of painters available: ";
    cin>>k;
    cout<<"Enter the lengths of "<<n<<" partitions : ";
    arr=(int*)malloc(n*(sizeof(int)));
    for(i=0; i<n; i++){
        cin>>arr[i];
    }
}
```

```

    }
    cout<<"-----";
    cout << endl<< "Minimum time to paint all boards: " << partition(arr, n, k)
<< endl;
    return 0;
}

```

OUTPUT:

 C:\Users\gv gbc\Documents\DAA lab\painter's partition recursive optimization.exe

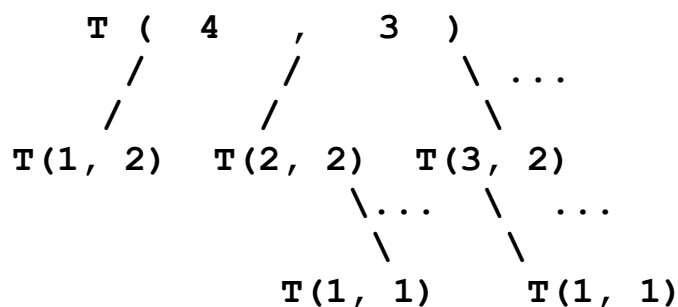
```

Enter the number of boards of varying length to be painted: 6
Enter the number of painters available: 3
Enter the lengths of 6 partitions : 10 40 20 30 40 50
-----
Minimum time to paint all boards: 70
-----
Process exited after 17.82 seconds with return value 0
Press any key to continue . . .

```

The Recursive Tree

Let's consider 4 boards and 3 painters



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.

Bottom Up Approach

Understanding bottom up approach in general:

Let's describe a state for our DP 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 i.e $dp[n]$.

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.

Code:

```
#include <climits>
#include <iostream>
#include <time.h>
using namespace std;

int sum(int arr[], int from, int to);
int findMax(int arr[], int n, int k);

// driver function
int main()
{
    int n, k, i;
    int *arr;
    float t;
    cout<<"Enter the number of partitions: ";
    cin>>n;
    cout<<"Enter the number of painters: ";
    cin>>k;
    cout<<"Enter the lengths of partitions: ";
    clock_t start,end;
    arr=(int*)malloc(n*(sizeof(int)));
    for(i=0; i<n; i++){
        cin>>arr[i];
    }
    cout<<"-----";
    cout<<endl<<"Minimum time: ";
    start = clock();
    cout<<findMax(arr, n, k)<<endl;
    end=clock();
```

```
cout<<"-----"<<endl;
t=(float) (((float) (end-start))/CLOCKS_PER_SEC);
cout<<"time taken: "<<t;
return 0;
}

// function to calculate sum between two indices
// in array
int sum(int arr[], int from, int to)
{
    int total = 0;
    for (int i = from; i <= to; i++)
        total += arr[i];
    return total;
}

// bottom up tabular dp
int findMax(int arr[], int n, int k)
{
    // initialize table
    int dp[k + 1][n + 1] = { 0 };
    // base cases
    // k=1
    for (int i = 1; i <= n; i++)
        dp[1][i] = sum(arr, 0, i - 1);
    // n=1
    for (int i = 1; i <= k; i++)
        dp[i][1] = arr[0];
    // 2 to k partitions
    for (int i = 2; i <= k; i++) { // 2 to n boards
        for (int j = 2; j <= n; j++) {
            // track minimum
            int best = INT_MAX;
            // i-1 th separator before position arr[p=1..j]
            for (int p = 1; p <= j; p++)
                best = min(best, max(dp[i - 1][p], sum(arr, p, j - 1)));
            dp[i][j] = best;
        }
    }
}
```

```
// required
return dp[k][n];
}
```

Output:

```
C:\Users\mahit\Documents\Academics\Assignments\DAA lab\seminar\bottom up dp.exe
Enter the number of partitions: 50
Enter the number of painters: 12
Enter the lengths of partitions:
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33
34 35 36 37 38 39 40 41 42 43 44 45 46 47 48 49 50
-----
Minimum time: 123
-----
time taken: 0.003
-----
Process exited after 51.18 seconds with return value 0
Press any key to continue . . .
```

Time Complexity:

Dynamic Programming Approach

Sum()

$$\text{Time complexity} = C_1 + C_2(n) + C_3(n-1) + C_4$$

$$= O(n)$$

FindMax()

$$\begin{aligned} \text{Time complexity} &= C_1 + C_2(n) + C_3(K) + C_4(K)^2 [C_5(n)^3 [C_6 + C_7(n) [C_8(n-1) + C_9]] + C_{10}] \\ &= C_1 + C_2 n + C_3 K + K \cdot C_4 [C_5^3 n + C_5^2 n^2 + C_5 n^3] \\ &= C_1 + C_2 n + C_3 K + C_a(Kn) + C_b K n^2 + C_c K n^3 \\ &= O(K n^3) \end{aligned}$$

$$T(n,k) = O(k \cdot n^3)$$

This can be easily brought down to $O(k \cdot n^2)$ by precomputing the cumulative sums in an array thus

avoiding repeated calls to the sum function.

Optimized Code:

```
#include <climits>
#include <iostream>
#include <time.h>
using namespace std;

int findMax(int arr[], int n, int k);

// driver function
int main() {
    int n, k, i;
    int *arr;

    cout<<"Enter the number of partitions: ";
    cin>>n;

    cout<<"Enter the number of painters: ";
    cin>>k;

    cout<<"Enter the lengths of partitions: ";
    arr=(int*)malloc(n*(sizeof(int)));

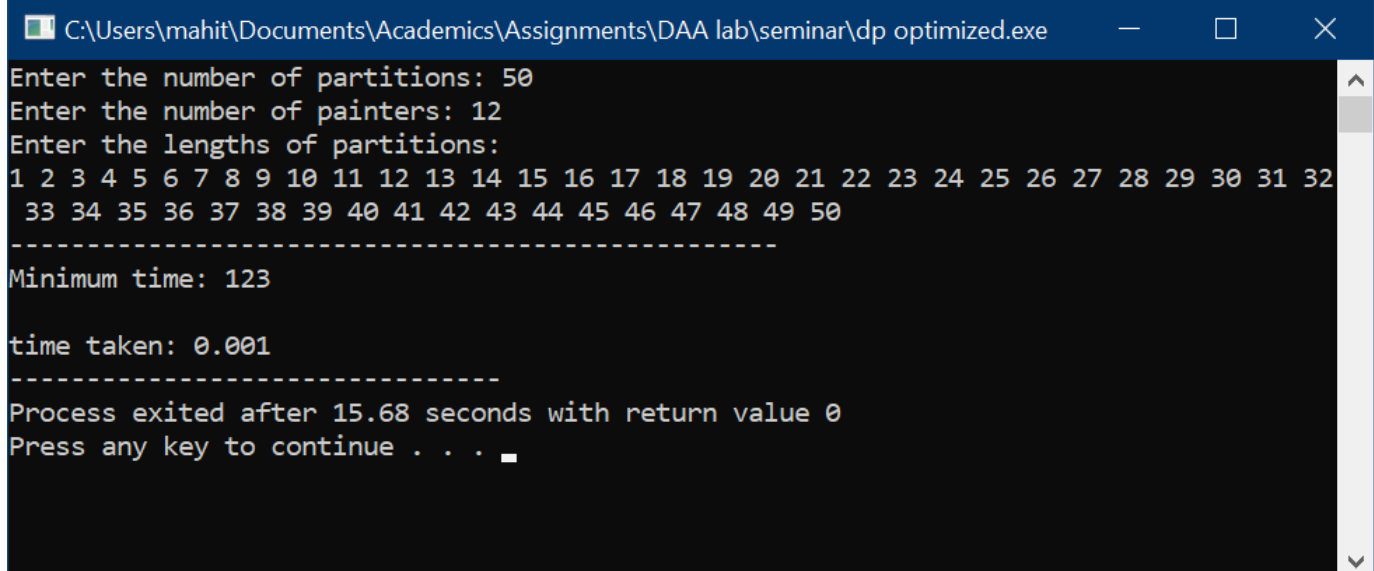
    for(i=0; i<n; i++){
        cin>>arr[i];
    }

    cout<<"-----";
    cout<<endl<<"Minimum time: ";
    clock_t start,end;
    float t;
    start=clock();
    cout<<findMax(arr, n, k) << endl;
    end=clock();
    t=(float) (((float) (end-start))/CLOCKS_PER_SEC);
    cout<<endl<<"time taken: "<<t;
    return 0;
}

// bottom up tabular dp
int findMax(int arr[], int n, int k){
    // initialize table
    int dp[k + 1][n + 1] = { 0 };
```

```
// base cases
// k=1
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];
// 2 to k partitions
for (int i = 2; i <= k; i++) { // 2 to n boards
    for (int j = 2; j <= n; j++) {
        // track minimum
        int best = INT_MAX;
        // i-1 th separator before position arr[p=1..j]
        for (int p = 1; p <= j; p++)
            best = min(best, max(dp[i-1][p], sum[j] - sum[p]));
        dp[i][j] = best;
    }
}
// required
return dp[k][n];
}
```

Output:



```
C:\Users\mahit\Documents\Academics\Assignments\DAA lab\seminar\dp optimized.exe
Enter the number of partitions: 50
Enter the number of painters: 12
Enter the lengths of partitions:
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32
33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48 49 50
-----
Minimum time: 123

time taken: 0.001
-----
Process exited after 15.68 seconds with return value 0
Press any key to continue . . .
```

Time Complexity:

Here, $T(n,k) = O(k \cdot n^2)$

Example

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

Answer = $DP[3][6] = 70$

Space Complexity:

Space Complexity of Bottom Up dynamic programming approach is $O(n \cdot k)$.

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

Binary search Approach

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 sections of array in the optimal allocation of boards.

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

How to find the range of our target?

Going by the understanding we have for the problem statement and by looking through the problem statements 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 and explanation for the same:

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. (i.e., 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 paint 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.

Note that x decreases while minimum time increases.

We want to find the minimum time under the constraint of $x = k$.

90	...	450
↑		↑
x=6		x=1

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

90	...	270	...	450
↑		↑		↑
x=6		x=2		x=1

Since $[2 < 3 \text{ so high}=270]$ we discard the upper half and continue searching in the lower half.

90	...	180	...	270
↑		↑		↑
x=6		x=3		x=2

[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.

90	...	135	...	180
↑		↑		↑
x=6		x=5		x=3

[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 the upper half [136,180].

After multiple successions of halving the search space, the final answer is 170, and its corresponding x is 3. This is also the minimum of time while maintaining the requirement $x = k$.

Observation

The lowest possible value of this range is the maximum value of the array max, as in this allocation we can allot max to one painter and divide the other sections such that the cost of them is less than or equal to max and as close as possible to max.

Now if we consider we use x painters in the above scenarios, it is obvious that as the value in the range increases, the value of x decreases and vice-versa. 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.

C++ Code

```
#include <iostream>

#include <bits/stdc++.h>

#include <time.h>
```

```
using namespace std;

clock_t begin, end;

double time_;

int getMax(int arr[], int n)
{
    int max = INT_MIN;

    for (int i = 0; i < n; i++)

        if (arr[i] > max)

            max = arr[i];

    return max;
}

int getSum(int arr[], int n)
{
    int sum = 0;

    for (int i = 0; i < n; i++)

        sum += arr[i];

    return sum;
}
```

```
int painter_count(int arr[], int n, int l) // find minimum required painters for
given maximim length l

{

    int total = 0, num = 1;

    for (int i = 0; i < n; i++)

    {

        total += arr[i];

        if (total > l)

        {

            total = arr[i];

            num++;

        }

    }

    return num;

}

int partition_range(int arr[], int n, int k)

{

    int low = getMax(arr, n);

    int high = getSum(arr, n);
```

```
while (low < high)

{

    int mid = low + (high - low) / 2;

    int req = painter_count(arr, n, mid); //required painter for the given
minimum time stored in req

    if (req <= k)

        high = mid;

    else

        low = mid + 1;

}

return low;

}

int main()

{

    int n, k, i;

    int *arr;

    cout << "Enter the number of boards of varying length to be painted: ";

    cin >> n;

    cout << "Enter the number of painters available: ";
```



```
cin >> k;

cout << "Enter the lengths of " << n << " partitions : ";

arr = (int *)malloc(n * (sizeof(int)));

for (i = 0; i < n; i++)

{

    cin >> arr[i];

}

cout << "-----";

//  cout << endl<< "Minimum time to paint all boards: " <<
partition_range(arr, n, k) << endl;

begin = clock();

cout << endl

    << "Minimum time to paint all boards: " << partition_range(arr, n, k) <<
endl;

end = clock();

time_ = ((double)(end - begin)) / CLOCKS_PER_SEC;

cout << "\nTime taken : " << time_;

return 0;

}
```

Result

```
D:\svn\sem4\daa\u19cs076_painter_binarysearch.exe
Enter the number of boards of varying length to be painted: 9
Enter the number of painters available: 3
Enter the lengths of 9 partitions : 1 2 3 4 5 6 7 8 9
-----
Minimum time to paint all boards: 17
Time taken :0
-----
Process exited after 5.272 seconds with return value 0
Press any key to continue . . .
```

```
D:\svn\sem4\daa\u19cs076_painter_binarysearch.exe
Enter the number of boards of varying length to be painted: 100
Enter the number of painters available: 72
Enter the lengths of 100 partitions : 377 839 732 447 496 29 640 921 381 443 79 537 44 758 134 689 592 271 195 73 275 20
7 298 920 945 451 953 478 40 402 529 176 179 452 981 75 463 407 164 879 605 585 730 253 764 455 438 881 992 405 996 797
325 923 474 48 328 851 486 612 994 633 32 580 810 234 370 649 746 512 710 397 414 50 846 827 718 968 30 332 56 887 715 9
89 604 379 229 52 972 390 380 26 910 173 655 469 199 232 160 190
-----
Minimum time to paint all boards: 996
Time taken :0.001
-----
Process exited after 5.888 seconds with return value 0
Press any key to continue . . .
```

Time and Space Complexity analysis

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

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

Binary - search approach

Painter_count()

$$\begin{aligned}\text{Time Complexity} &= C_1 + C_2(n) + C_3(n-1) + C_4(n-1) + C_5(n-1) + C_6(n-1) + C_7 \\ &= A(n) + B \\ &= O(n)\end{aligned}$$

Partition_range()

Time Complexity \Rightarrow

The size of search range is from low to high

According to binary search algorithm, (invariant) painter-counted is called $\log(\text{high})$ times upper bound

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

The time complexity of the above approach is $O(n \times \log(\text{sum(arr[])}))$.

Space complexity is $O(1)$

APPLICATION 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.

CONCLUSION

Having successfully solved the Painter's Partition problem, we have learnt that the Binary search method has proved to be the best approach with least time and space complexity.

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 because we noticed some repeated recursive calls, we used the bottom up approach and increased the efficiency by bringing time complexity to $O(k \cdot N^3)$ but it increased the space complexity to $O(k \cdot N)$ extra space. After trying our modifications in dynamic code we could bring it to $O(k \cdot N^2)$.

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 where high is sum of all length of boards and low is the maximum length of board to be painted. Now we perform binary search and 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. This proved to be the most optimum approach with lesser space and time complexity. The time complexity was thus brought down to $O(n \cdot \log(\text{sum}(\text{arr}[])))$.

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.