CONTENT

Problem Definition

There are n people and two identical voting machines. We are also given an array a[] of size n such that a[i] stores time required by i-th person to go to any machine, mark his vote and come back. At one time instant, only one person can be there on each of the machines. Given a value x, defining the maximum allowable time for which machines are operational, check whether all persons can cast their vote or not.

Problem Explanation with diagram and example

Let **sum** be the total time taken by all n people. If sum <=x, then answer will obviously be YES. Otherwise, we need to check whether the given array can be split into two parts such that the sum of the first part and sum of the second part are both less than or equal to x. The problem is similar to the knapsack problem. Imagine two knapsacks each with capacity x. Now find, maximum people who can vote on any one machine i.e. find maximum subset sum for a knapsack of capacity x. Let this sum be s1. Now if (sum-s1) <= x, then answer is YES else answer is NO.

Code

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

// Returns true if n people can vote using
// two machines in x time.
bool canVote(int a[], int n, int x)
{
    // dp[i][j] stores maximum possible number
    // of people among arr[0..i-1] can vote
    // in j time.
    int dp[n+1][x+1];
    memset(dp, 0, sizeof(dp));

    // Find sum of all times
    int sum = 0;
    for (int i=0; i<=n; i++ )
        sum += a[i];

    // Fill dp[][] in bottom up manner (Similar
    // to knapsack).
    for (int i=1; i<=n; i++)
        for (int j=1; j<=x; j++)
```

```
            if (a[i] <= j)
                dp[i][j] = max(dp[i-1][j],
                        a[i] + dp[i-1][j-a[i]]);
            else
                dp[i][j] = dp[i-1][j];

    return (sum - dp[n][x] <= x);
}

// Driver code
int main()
{
    int n = 3, x = 4;
    int a[] = {2, 4, 2};
    canVote(a, n, x)? cout << "YES\n" :
                      cout << "NO\n";
    return 0;
}
```

Example

Let's understand the problem in following steps-

There N people in line to vote.

There are 2 voting machines available. These machines will only be up and running for X minutes.

If the i*th* person takes a[i] minutes to vote, then we need to find out if all people can vote on 2 machines within the time X.

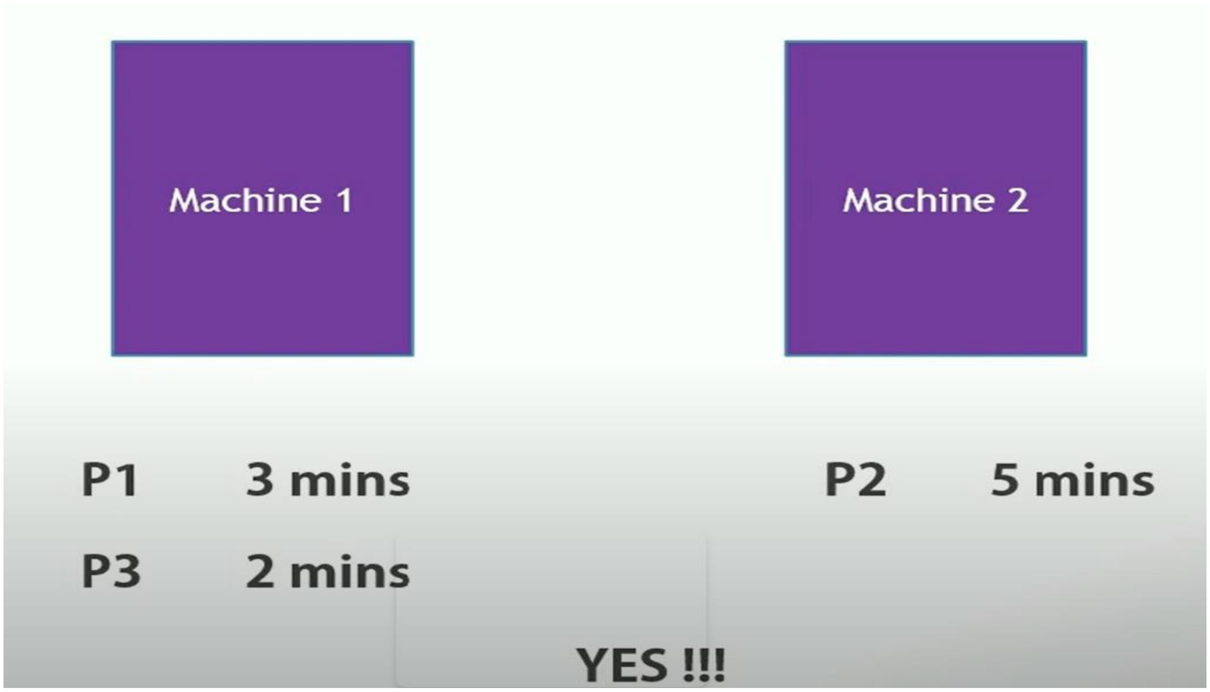
Here's the data -

N=3

X= 5 mins

T= 2,5,3

So, there are 3 people to vote. P1 takes 2 mins, P2 takes 5 mins and P3 takes 3 mins.


Machine 1
Machine 2
P1 3 mins
P2 5 mins
P3 2 mins
YES !!!

Design Techniques Used

We used Dynamic Programming in this concept.

Dynamic Programming

Dynamic Programming is the most powerful design technique for solving optimization problems.

Dynamic Programming is used when the subproblems are not independent, e.g., when they share the same subproblems. In this case, divide and conquer may do more work than necessary, because it solves the same sub problem multiple times.

Dynamic Programming is a Bottom-up approach- we solve all possible small problems and then combine to obtain solutions for bigger problems.

Dynamic Programming is a paradigm of algorithm design in which an optimization problem is solved by a combination of achieving sub-problem solutions and appearing to the "principle of optimality".

Characteristics of Dynamic Programming

Dynamic Programming works when a problem has the following features: -

•Optimal Substructure: If an optimal solution contains optimal sub solutions, then a problem exhibits optimal substructure.

•Overlapping sub-problems: When a recursive algorithm would visit the same sub-problems repeatedly, then a problem has overlapping sub-problems.

Applications of Dynamic Programming

1. 0/1 knapsack problem
2. Mathematical optimization problem
3. All pair shortest path problem
4. Reliability design problem
5. Longest common subsequence (LCS)

This DP problem is based on the 01-knapsack problem. The plot of this problem is intriguing and it is evident from its title only.

The 0/1 knapsack problem

The 0/1 knapsack problem means that the items are either completely or no items are filled in a knapsack. For example, we have two items having weights 2kg and 3kg, respectively. If we pick the 2kg item then we cannot pick 1kg item from the 2kg item (item is not divisible); we have to pick the 2kg item completely. This is a 0/1 knapsack problem in which either we pick the item completely or we will pick that item. The 0/1 knapsack problem is solved by the dynamic programming.

0/1 Knapsack Complexity

Time Complexity: O(N*W).
where 'N' is the number of weight element and 'W' is capacity. As for

every weight element we traverse through all weight capacities 1<=w<=W.

Implementation of 0/1knapsack for the given problem statement

		1	2	3	4	5
	0	0	0	0	0	0
3	0	0	0	3	3	3
5	0	0	0	3	3	5
2	0	0	2	3	3	5

Algorithm

```
//sum of all times
for i:=0 to n do
      sum := sum + a[i];
//fill dp[] in bottom up manner (similar to knapsack)
for i:=1 to n do
      for j:=1 to x do
           if (a[i] <= j) then
                dp[i][j] := max(dp[i-1][j], a[i] + dp[i-1][j-a[i]]);
                //dp[i][j] stores maximum possible number of people
                //who can vote in j time
             else
                   dp[i][j] :=dp[i-1][j];
return (sum - dp[n][x] <= x);
```

Explanation of Algorithm

Here, the sum is the sum of the time taken by n people. The array of people is denoted by a[i]. The maximum number of people among the array that can vote in j time is stored in dp[i][j].

Example

Input: n = 3, x = 4, a[] = {2, 4, 2}

There are n = 3 persons say and maximum allowed time is x = 4 units. Let the persons be P0, P1, and P2 and the two machines be M0 and M1. The sum is 8 units.

At t0: P0 goes to M0

At t0: P2 goes to M1

At t2: M0 is free, P3 goes to M0

At t4: both M0 and M1 are free and all 3 have given their vote.

Therefore, all n people can cast their vote at the voting machines.

Output: YES

Complexity Analysis

```
for (int i=1; i<=n; i++)
        for (int j=1; j<=x; j++)
            if (a[i] <= j)
                dp[i][j] = max(dp[i-1][j],
                        a[i] + dp[i-1][j-a[i]]);
            else
                dp[i][j] = dp[i-1][j];

    return (sum - dp[n][x] <= x);
```

*Time Complexity: O(x*n)*

Conclusion

The problem of casting a vote is solved using the dynamic programming (0/1knapsack) method and an example is provided with the algorithm and its explanation. This method can be used to check whether all people(voters) can cast their vote or not in the given two machines within a given time constraint.

References

https://www.youtube.com/watch?v=liIlrM57xy4&ab_channel=Joey%27sTech

https://www.programiz.com/dsa/dynamic-programming#:~:text=Dynamic%20Programming%20is%20a%20technique,subproblems%20and%20optimal%20substructure%20property.

https://www.geeksforgeeks.org/0-1-knapsack-problem-dp-10/

https://en.wikipedia.org/wiki/Dynamic_programming