Given an unsorted array, trim the array such that twice of minimum is greater than maximum in the trimmed array. Elements should be removed either end of the array.

Number of removals should be minimum.

Examples:

```
arr[] = {4, 5, 100, 9, 10, 11, 12, 15, 200}
Output: 4
We need to remove 4 elements (4, 5, 100, 200)
so that 2*min becomes more than max.


arr[] = {4, 7, 5, 6}
Output: 0
We don't need to remove any element as
4*2 > 7 (Note that min = 4, max = 8)

arr[] = {20, 7, 5, 6}
Output: 1
We need to remove 20 so that 2*min becomes
more than max

arr[] = {20, 4, 1, 3}
Output: 3
We need to remove any three elements from ends
like 20, 4, 1 or 4, 1, 3 or 20, 3, 1 or 20, 4, 1
```

Naive Solution:
A naive solution is to try every possible case using recurrence. Following is the naive recursive algorithm. Note that the algorithm only returns minimum numbers of removals to be made, it doesn't print the trimmed array. It can be easily modified to print the trimmed array as well.

```
// Returns minimum number of removals to be made in
// arr[l..h]
minRemovals(int arr[], int l, int h)
1) Find min and max in arr[l..h]
2) If 2*min > max, then return 0.
3) Else return minimum of "minRemovals(arr, l+1, h) + 1"
   and "minRemovals(arr, l, h-1) + 1"
```

Following is C++ implementation of above algorithm.

```
#include <iostream>
using namespace std;

// A utility function to find minimum of two numbers
int min(int a, int b) {return (a < b)? a : b;}

// A utility function to find minimum in arr[l..h]
int min(int arr[], int l, int h)
{
    int mn = arr[l];
    for (int i=l+1; i<=h; i++)
       if (mn > arr[i])
         mn = arr[i];
```

```
    return mn;
}

// A utility function to find maximum in arr[l..h]
int max(int arr[], int l, int h)
{
    int mx = arr[l];
    for (int i=l+1; i<=h; i++)
       if (mx < arr[i])
         mx = arr[i];
    return mx;
}

// Returns the minimum number of removals from either end
// in arr[l..h] so that 2*min becomes greater than max.
int minRemovals(int arr[], int l, int h)
{
    // If there is 1 or less elements, return 0
    // For a single element, 2*min > max
    // (Assumption: All elements are positive in arr[])
    if (l >= h) return 0;

    // 1) Find minimum and maximum in arr[l..h]
    int mn = min(arr, l, h);
    int mx = max(arr, l, h);

    //If the property is followed, no removals needed
    if (2*mn > mx)
       return 0;

    // Otherwise remove a character from left end and recur,
    // then remove a character from right end and recur, take
    // the minimum of two is returned
    return min(minRemovals(arr, l+1, h),
               minRemovals(arr, l, h-1)) + 1;
}

// Driver program to test above functions
int main()
{
  int arr[] = {4, 5, 100, 9, 10, 11, 12, 15, 200};
  int n = sizeof(arr)/sizeof(arr[0]);
  cout << minRemovals(arr, 0, n-1);
  return 0;
}
```

Output:

```
4
```

Time complexity: Time complexity of the above function can be written as following

```
T(n) = 2T(n-1) + O(n)
```

An upper bound on solution of above recurrence would be $O(n \times 2^n)$.

Dynamic Programming:
The above recursive code exhibits many overlapping subproblems. For example minRemovals(arr, l+1, h-1) is evaluated twice. So Dynamic Programming is the choice to optimize the solution. Following is Dynamic Programming based solution.

```
#include <iostream>
using namespace std;

// A utility function to find minimum of two numbers
int min(int a, int b) {return (a < b)? a : b;}
```

```
// A utility function to find minimum in arr[l..h]
int min(int arr[], int l, int h)
{
    int mn = arr[l];
    for (int i=l+1; i<=h; i++)
       if (mn > arr[i])
         mn = arr[i];
    return mn;
}

// A utility function to find maximum in arr[l..h]
int max(int arr[], int l, int h)
{
    int mx = arr[l];
    for (int i=l+1; i<=h; i++)
       if (mx < arr[i])
         mx = arr[i];
    return mx;
}

// Returns the minimum number of removals from either end
// in arr[l..h] so that 2*min becomes greater than max.
int minRemovalsDP(int arr[], int n)
{
    // Create a table to store solutions of subproblems
    int table[n][n], gap, i, j, mn, mx;

    // Fill table using above recursive formula. Note that the table
    // is filled in diagonal fashion (similar to http://goo.gl/PQqoS),
    // from diagonal elements to table[0][n-1] which is the result.
    for (gap = 0; gap < n; ++gap)
    {
        for (i = 0, j = gap; j < n; ++i, ++j)
        {
            mn = min(arr, i, j);
            mx = max(arr, i, j);
            table[i][j] = (2*mn > mx)? 0: min(table[i][j-1]+1,
                                              table[i+1][j]+1);
        }
    }
    return table[0][n-1];
}

// Driver program to test above functions
int main()
{
  int arr[] = {20, 4, 1, 3};
  int n = sizeof(arr)/sizeof(arr[0]);
  cout << minRemovalsDP(arr, n);
  return 0;
}
```

Time Complexity: $O(n^3)$ where n is the number of elements in arr[].

Further Optimizations:
The above code can be optimized in many ways.
1) We can avoid calculation of min() and/or max() when min and/or max is/are not changed by removing corner elements.

2) We can pre-process the array and build segment tree in O(n) time. After the segment tree is built, we can query range minimum and maximum in O(Logn) time. The overall time complexity is reduced to $O(n^2Logn)$ time.

A O(n^2) Solution
The idea is to find the maximum sized subarray such that 2*min > max. We run two nested loops, the outer loop chooses a starting point and the inner loop chooses ending point for the current starting point. We keep track of

longest subarray with the given property.

Following is C++ implementation of the above approach. Thanks to Richard Zhang for suggesting this solution.

```
// A O(n*n) solution to find the minimum of elements to
// be removed
#include <iostream>
#include <climits>
using namespace std;

// Returns the minimum number of removals from either end
// in arr[l..h] so that 2*min becomes greater than max.
int minRemovalsDP(int arr[], int n)
{
    // Initialize starting and ending indexes of the maximum
    // sized subarray with property 2*min > max
    int longest_start = -1, longest_end = 0;

    // Choose different elements as starting point
    for (int start=0; start<n; start++)
    {
        // Initialize min and max for the current start
        int min = INT_MAX, max = INT_MIN;

        // Choose different ending points for current start
        for (int end = start; end < n; end ++)
        {
            // Update min and max if necessary
            int val = arr[end];
            if (val < min) min = val;
            if (val > max) max = val;

            // If the property is violated, then no
            // point to continue for a bigger array
            if (2 * min <= max) break;

            // Update longest_start and longest_end if needed
            if (end - start > longest_end - longest_start ||
                longest_start == -1)
            {
                longest_start = start;
                longest_end = end;
            }
        }
    }

    // If not even a single element follow the property,
    // then return n
    if (longest_start == -1) return n;

    // Return the number of elements to be removed
    return (n - (longest_end - longest_start + 1));
}

// Driver program to test above functions
int main()
{
    int arr[] = {4, 5, 100, 9, 10, 11, 12, 15, 200};
    int n = sizeof(arr)/sizeof(arr[0]);
    cout << minRemovalsDP(arr, n);
    return 0;
}
```