Bin Packing Problem (Minimize number of used Bins)

Given n items of different weights and bins each of capacity c, assign each item to a bin such that number of total used bins is minimized. It may be assumed that all items have weights smaller than bin capacity.

Example:

```
Input:  weight[]       = {4, 8, 1, 4, 2, 1}
        Bin Capacity c = 10
Output: 2
We need minimum 2 bins to accommodate all items
First bin contains {4, 4, 2} and second bin {8, 1, 1}

Input:  weight[]       = {9, 8, 2, 2, 5, 4}
        Bin Capacity c = 10
Output: 4
We need minimum 4 bins to accommodate all items.

Input:  weight[]       = {2, 5, 4, 7, 1, 3, 8};
        Bin Capacity c = 10
Output: 3
```

Lower Bound

We can always find a lower bound on minimum number of bins required. The lower bound can be given as :

```
Min no. of bins  >=  Ceil ((Total Weight) / (Bin Capacity))
```

In the above examples, lower bound for first example is “ceil(4 + 8 + 1 + 4 + 2 + 1)/10” = 2 and lower bound in second example is “ceil(9 + 8 + 2 + 2 + 5 + 4)/10” = 3.
This problem is a NP Hard problem and finding an exact minimum number of bins takes exponential time. Following are approximate algorithms for this problem.

Applications

1. Loading of containers like trucks.
2. Placing data on multiple disks.
3. Job scheduling.
4. Packing advertisements in fixed length radio/TV station breaks.
5. Storing a large collection of music onto tapes/CD’s, etc.

Online Algorithms

These algorithms are for Bin Packing problems where items arrive one at a time (in unknown order), each must be put in a bin, before considering the next item.

1. Next Fit:

When processing next item, check if it fits in the same bin as the last item. Use a new bin only if it does not.
Below is C++ implementation for this algorithm.

C++ Java Python3 C# Javascript

```cpp
// C++ program to find number of bins required using
// next fit algorithm.
#include <bits/stdc++.h>
using namespace std;

// Returns number of bins required using next fit
// online algorithm
int nextFit(int weight[], int n, int c)
{
    // Initialize result (Count of bins) and remaining
    // capacity in current bin.
    int res = 0, bin_rem = c;

    // Place items one by one
    for (int i = 0; i < n; i++) {
        // If this item can't fit in current bin
        if (weight[i] > bin_rem) {
            res++; // Use a new bin
            bin_rem = c - weight[i];
        }
        else
            bin_rem -= weight[i];
    }
    return res;
}

// Driver program
int main()
{
    int weight[] = { 2, 5, 4, 7, 1, 3, 8 };
    int c = 10;
    int n = sizeof(weight) / sizeof(weight[0]);
    cout << "Number of bins required in Next Fit : "
         << nextFit(weight, n, c);
    return 0;
}
```

Output:

```
Number of bins required in Next Fit : 4
```

Next Fit is a simple algorithm. It requires only O(n) time and O(1) extra space to process n items.
Next Fit is 2 approximate, i.e., the number of bins used by this algorithm is bounded by twice of optimal. Consider any two adjacent bins. The sum of items in these two bins must be > c; otherwise, NextFit would have put all the items of second bin into the first. The same holds for all other bins. Thus, at most half the space is wasted, and so Next Fit uses at most 2M bins if M is optimal.

2. First Fit:

When processing the next item, scan the previous bins in order and place the item in the first bin that fits. Start a new bin only if it does not fit in any of the existing bins.

C++ Java Python3 C# Javascript

```cpp
// C++ program to find number of bins required using
// First Fit algorithm.
#include <bits/stdc++.h>
using namespace std;

// Returns number of bins required using first fit
// online algorithm
int firstFit(int weight[], int n, int c)
{
    // Initialize result (Count of bins)
    int res = 0;

    // Create an array to store remaining space in bins
    // there can be at most n bins
    int bin_rem[n];

    // Place items one by one
    for (int i = 0; i < n; i++) {
        // Find the first bin that can accommodate
        // weight[i]
        int j;
        for (j = 0; j < res; j++) {
            if (bin_rem[j] >= weight[i]) {
                bin_rem[j] = bin_rem[j] - weight[i];

                break;
            }
        }

        // If no bin could accommodate weight[i]
        if (j == res) {
            bin_rem[res] = c - weight[i];
            res++;
        }

    }
    return res;
}

// Driver program
int main()
{
    int weight[] = { 2, 5, 4, 7, 1, 3, 8 };
    int c = 10;
    int n = sizeof(weight) / sizeof(weight[0]);
    cout << "Number of bins required in First Fit : "
         << firstFit(weight, n, c);
    return 0;
}
```

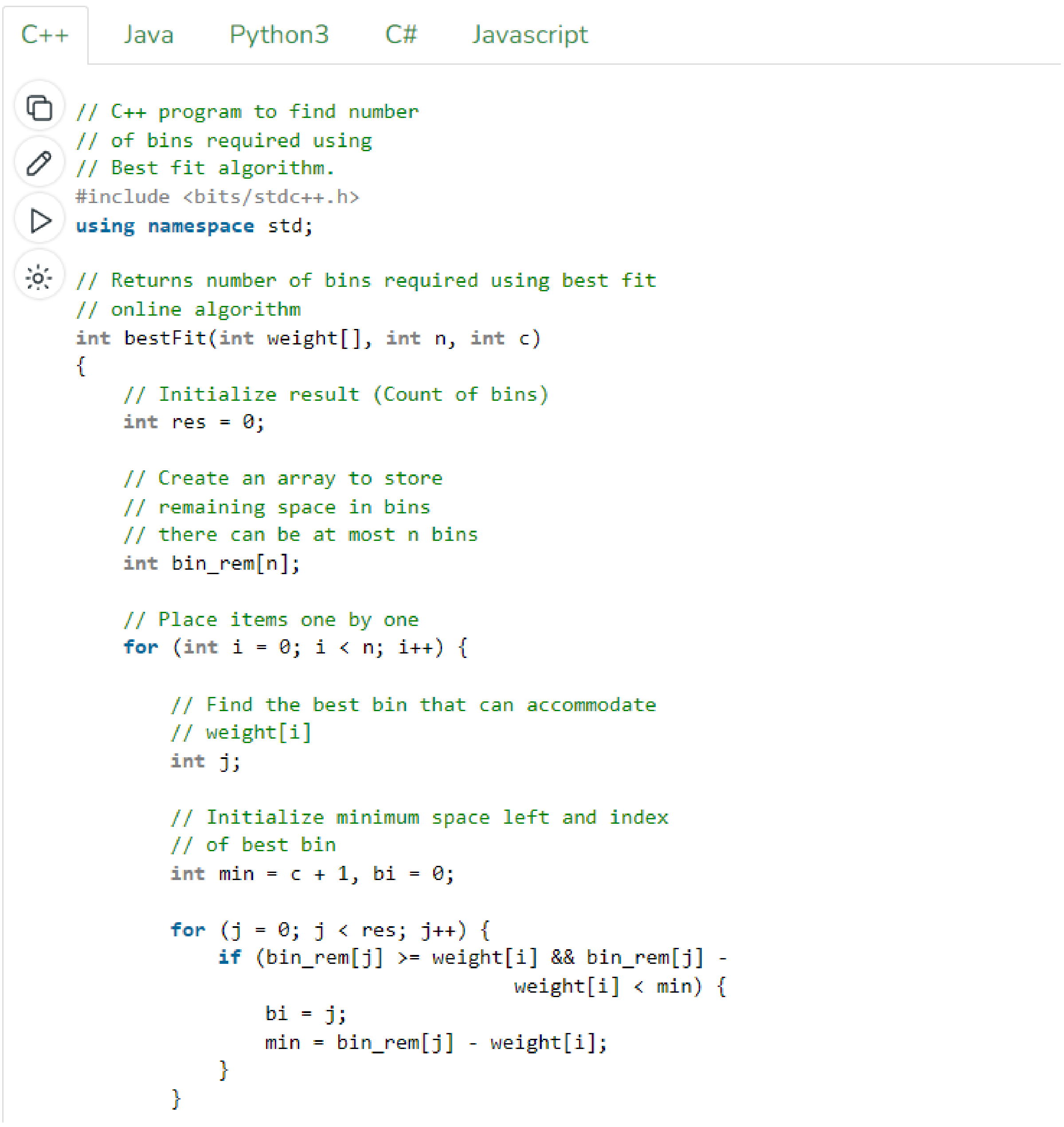
Output:

```
Number of bins required in First Fit : 4
```

The above implementation of First Fit requires O(n^2) time, but First Fit can be implemented in O(n Log n) time using Self-Balancing Binary Search Trees.
If M is the optimal number of bins, then First Fit never uses more than 1.7M bins. So First-Fit is better than Next Fit in terms of upper bound on number of bins.

Auxiliary Space: O(n)

3. Best Fit:
The idea is to places the next item in the *tightest* spot. That is, put it in the bin so that the smallest empty space is left.



```cpp
// C++ program to find number
// of bins required using
// Best fit algorithm.
#include <bits/stdc++.h>
using namespace std;

// Returns number of bins required using best fit
// online algorithm
int bestFit(int weight[], int n, int c)
{
    // Initialize result (Count of bins)
    int res = 0;

    // Create an array to store
    // remaining space in bins
    // there can be at most n bins
    int bin_rem[n];

    // Place items one by one
    for (int i = 0; i < n; i++) {

        // Find the best bin that can accommodate
        // weight[i]
        int j;

        // Initialize minimum space left and index
        // of best bin
        int min = c + 1, bi = 0;

        for (j = 0; j < res; j++) {
            if (bin_rem[j] >= weight[i] && bin_rem[j] -
                                      weight[i] < min) {
                bi = j;
                min = bin_rem[j] - weight[i];
            }
        }
        // If no bin could accommodate weight[i],
        // create a new bin
        if (min == c + 1) {
            bin_rem[res] = c - weight[i];
            res++;
        }
        else // Assign the item to best bin
            bin_rem[bi] -= weight[i];
    }
    return res;
}

// Driver program
int main()
{
    int weight[] = { 2, 5, 4, 7, 1, 3, 8 };
    int c = 10;
    int n = sizeof(weight) / sizeof(weight[0]);
    cout << "Number of bins required in Best Fit : "
         << bestFit(weight, n, c);
    return 0;
}
```

Output:

```
Number of bins required in Best Fit : 4
```

Best Fit can also be implemented in O(n Log n) time using Self-Balancing Binary Search Trees.
If M is the optimal number of bins, then Best Fit never uses more than 1.7M bins. So Best Fit is same as First Fit and better than Next Fit in terms of upper bound on number of bins.

Auxiliary Space: O(n)

4. Worst Fit:

The idea is to places the next item in the least tight spot to even out the bins. That is, put it in the bin so that most empty space is left.

C++ Java Python3 C# Javascript

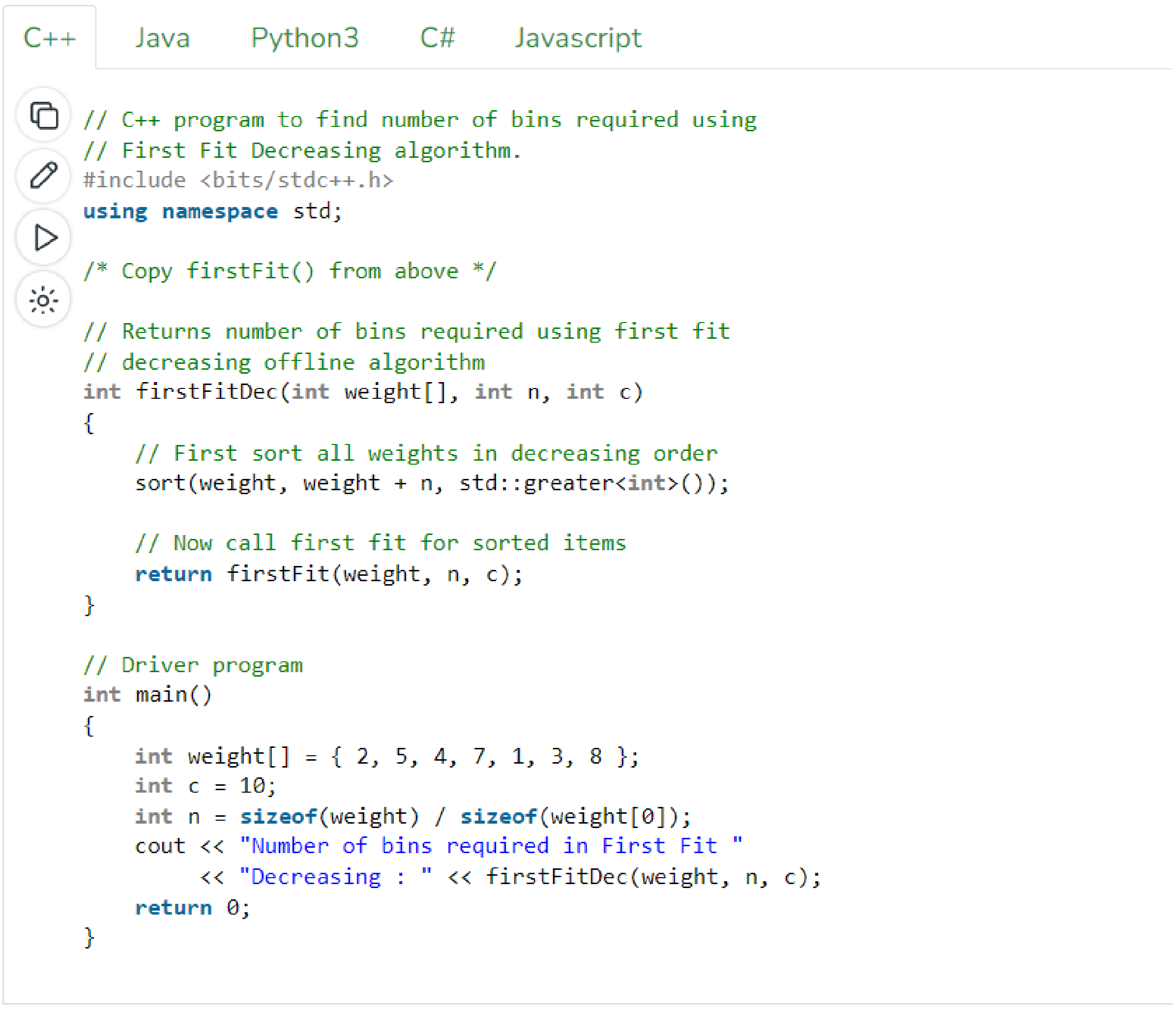
```
// C++ program to find number of bins required using
// Worst fit algorithm.
#include <bits/stdc++.h>
using namespace std;

// Returns number of bins required using worst fit
// online algorithm
int worstFit(int weight[], int n, int c)
{
    // Initialize result (Count of bins)
    int res = 0;

    // Create an array to store remaining space in bins
    // there can be at most n bins
    int bin_rem[n];

    // Place items one by one
    for (int i = 0; i < n; i++) {
        // Find the best bin that can accommodate
        // weight[i]
        int j;

        // Initialize maximum space left and index
        // of worst bin
        int mx = -1, wi = 0;
        for (j = 0; j < res; j++) {
            if (bin_rem[j] >= weight[i] && bin_rem[j] - weight[i] > mx) {
                wi = j;
                mx = bin_rem[j] - weight[i];
            }
        }

        // If no bin could accommodate weight[i],
        // create a new bin
        if (mx == -1) {
            bin_rem[res] = c - weight[i];
            res++;
        }
        else // Assign the item to best bin
            bin_rem[wi] -= weight[i];
    }
    return res;
}

// Driver program
int main()
{
    int weight[] = { 2, 5, 4, 7, 1, 3, 8 };
    int c = 10;
    int n = sizeof(weight) / sizeof(weight[0]);
    cout << "Number of bins required in Worst Fit : "
         << worstFit(weight, n, c);
    return 0;
}

// This code is contributed by gromperen
```

Output:

```
Number of bins required in Worst Fit : 4
```

Worst Fit can also be implemented in O(n Log n) time using Self-Balancing Binary Search Trees.
If M is the optimal number of bins, then Best Fit never uses more than 2M-2 bins. So Worst Fit is same as Next Fit in terms of upper bound on number of bins.
Auxiliary Space: O(n)

Offline Algorithms
In the offline version, we have all items upfront. Unfortunately offline version is also NP Complete, but we have a better approximate algorithm for it. First Fit Decreasing uses at most (4M + 1)/3 bins if the optimal is M.
4. First Fit Decreasing:
A trouble with online algorithms is that packing large items is difficult, especially if they occur late in the sequence. We can circumvent this by *sorting* the input sequence, and placing the large items first. With sorting, we get First Fit Decreasing and Best Fit Decreasing, as offline analogues of online First Fit and Best Fit.



```cpp
// C++ program to find number of bins required using
// First Fit Decreasing algorithm.
#include <bits/stdc++.h>
using namespace std;

/* Copy firstFit() from above */

// Returns number of bins required using first fit
// decreasing offline algorithm
int firstFitDec(int weight[], int n, int c)
{
    // First sort all weights in decreasing order
    sort(weight, weight + n, std::greater<int>());

    // Now call first fit for sorted items
    return firstFit(weight, n, c);
}

// Driver program
int main()
{
    int weight[] = { 2, 5, 4, 7, 1, 3, 8 };
    int c = 10;
    int n = sizeof(weight) / sizeof(weight[0]);
    cout << "Number of bins required in First Fit "
         << "Decreasing : " << firstFitDec(weight, n, c);
    return 0;
}
```

Output:

```
Number of bins required in First Fit Decreasing : 3
```

First Fit decreasing produces the best result for the sample input because items are sorted first. First Fit Decreasing can also be implemented in O(n Log n) time using Self-Balancing Binary Search Trees.

Auxiliary Space: O(1)

This article is contributed by **Dheeraj Gupta**. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above