July 16, 2025

CLASSICAL GREEDY PROBLEMS

* Greedy Technique - In order to solve a problem, always take the best possible choice. Usually solvable in $O(n)$, $O(\log n)$ time complexity with constraints like: $n \to 10^5, 10^6$.

When we have to take all possible choices as consideration then that's a dynamic programming problem. Usually solvable in n^2, n^3, $n\log n$ T.C. with constraints ranging like $n \to 10^3$.

#1. sum = 15, denomination of coins given → 1/2

form this sum using given denomination (min)

The most intuitive approach is usually greedy. Ex:

$C_1 = 1$

$C_2 = 2$

Use the largest amount to form however much of the sum $2 * 7 = 14$

1 is remaining, we will use C_1 to counter the difference.

Hence,

$$(2 * 7) + (1 * 1) = 15$$

Min. no. of coins used = 8 (7 coins of C_2, 1 of C_1) to form the sum

Sometimes, it can be a little non-trivial as well. Let us suppose we want to form the sum = 16 using coins of 3/2. If we were to approach this problem using our above one then we would want to maximise usage of 3 but $3 * 5 = 15$ and no other combination of coin can be used to form 16. We shall observe that

in such case there is some permutation/combination is coming into play to determine how many times a coin will be used to form the sum. In this case, sum = 16,

$(3*4) + (2*2) = 16$

In such cases, dynamic programming is used. In essence, Greedy Technique can be called a subset of DP.

↳ we settled earlier before reaching the max value < sum that can be formed using coins of 3.

* Important pattern that can be observed in Greedy problems is that when it is applied to a collection of items like array, string then SORTING will always be a given.

* In problems where a range is given and is used to determine the final answer then Greedy Technique is used in such problems.

#2 Let us suppose there are a list of jobs spanning over a start point till an ending point

→ 1	5
→ 1	3
→ 1	2
→ 4	7
→ 2	4
→ 3	6

Find the maximum number of non-overlapping jobs the worker can attend. A single worker would be attending jobs one by one.

{Higher constraint} → $10^5/10^6$ → Greedy

If always taking the Best choice benefits

Approach: Earlier job is finished, more choices as remaining time can be used to do more jobs.

Suppose you choose (1, 2) you will be done at 2 but if you take (1, 5) you end later as it has a broader scope hence it may also overlap. Therefore, we want to try pick shorter intervals to avoid the problem of overlapping and maximising the job. So, in order to achieve this, Let us sort by end times.

After sort: (1, 2) (1, 3) (2, 4) (1, 5) (3, 6) (4, 7)

1 2 ✓ cnt = 1, end = 2

1 3 ✗ Can this be done with prev job? Nope, since it is overlapping

2 4 ✓ No overlap, cnt = 2, end = 4.

1 5 ✗ overlap

3 6 ✗ overlap

4 7 ✓ cnt = 3, end = 7 Ans

Time complexity: O(N)
nlogn for sorting
N iteration
⇒ O(nlogn + n)

Problem: https://leetcode.com/problems/non-overlapping-intervals/description/

435. Non-overlapping Intervals

Medium | Topics | Companies

Given an array of intervals `intervals` where `intervals[i] = [start_i, end_i]`, return *the minimum number of intervals you need to remove to make the rest of the intervals non-overlapping.*

Note that intervals which only touch at a point are **non-overlapping**. For example, `[1, 2]` and `[2, 3]` are non-overlapping.

```cpp
int eraseOverlapIntervals(vector<vector<int>>& intervals) {

    sort(intervals.begin(), intervals.end(), [](const vector<int>& a, const vector<int>& b){
        return a[1] < b[1];
    });

    int count = 0;
    int end = INT_MIN;

    for(auto interval : intervals){

        if(interval[0] < end){
            count++;
        }
        else{
            end = interval[1];
        }
    }
```

← sort by end times

// Non-Overlapping

Problem: https://www.geeksforgeeks.org/dsa/fractional-knapsack-problem/

Fractional Knapsack Problem

Last Updated : 20 Apr, 2025

Given two arrays, **val[]** and **wt[]**, representing the values and weights of items, and an integer **capacity** representing the maximum weight a knapsack can hold, the task is to determine the **maximum total value** that can be achieved by putting items in the knapsack. You are allowed to break items into fractions if necessary.

Note: Return the maximum value as a double, rounded to 6 decimal places.

Given:
weight of elements, w[] = 10, 20, 30
value of elements, v[] = 60, 100, 120
cap = 50

→ Either take an element fully or don't take → 0/1 (typical knapsack problem)

But Here for fractional Knapsack,
For an element e, it can be taken as $\frac{1}{4}, \frac{1}{2}, \frac{1}{10}, \frac{1}{8}$ etc.

Approach:
Elements having highest ratio $\left(\frac{val}{wt}\right)$ will be preferred

Ratio of above given example: 6, 5, 4 [60]
↳ 0th index element has highest ratio

Knapsack capacity = 50, acc. to highest ratio, can we put wt[0] = 10 in Knapsack? Yes as 10 < 50, so new capacity = 40

→ ratio = 5, wt[1] = 20

Can wt[1] be inserted in our knapsack with capacity = 40.

Yes. [60 + 100]

New capacity = 20

→ ratio = 4, wt[2] = 30

30 > New capacity

So we can't take whole of wt[2]. But we know this is fractional knapsack. hence we can just take 20 from wt[2] = 30 to fill the capacity.

→ How much val are we taking for wt[2] = 20.

20 → 4 * 20

= 80. [60 + 100 + 80]

Ans 240

Step by step approach:

1. Calculate the ratio (**profit/weight**) for each item.
2. Sort all the items in decreasing order of the ratio.
3. Initialize **res = 0**, current capacity= given capacity.
4. Do the following for every item **i** in the sorted order:
   - If the weight of the current item is less than or equal to the remaining capacity then add the value of that item into the result
   - Else add the current item as much as we can and break out of the loop.
5. Return **res**.

Time Complexity : O(N)

n ratios → sort → O(nlogn)

n iterations for Knapsack calc → O(n)

⇒ O(nlogn + n)

⇒ O(n)

Soln

```cpp
// Comparison function to sort items based on value/weight ratio
bool compare(vector<int>& a, vector<int>& b) {
    double a1 = (1.0 * a[0]) / a[1];
    double b1 = (1.0 * b[0]) / b[1];
    return a1 > b1;
}

double fractionalKnapsack(vector<int>& val, vector<int>& wt, int capacity) {
    int n = val.size();

    // Create 2D vector to store value and weight
    // items[i][0] = value, items[i][1] = weight
    vector<vector<int>> items(n, vector<int>(2));

    for (int i = 0; i < n; i++) {
        items[i][0] = val[i];
        items[i][1] = wt[i];
    }

    // Sort items based on value-to-weight ratio in descending order
    sort(items.begin(), items.end(), compare);

    double res = 0.0;
    int currentCapacity = capacity;

    // Process items in sorted order
    for (int i = 0; i < n; i++) {

        // If we can take the entire item
        if (items[i][1] <= currentCapacity) {
            res += items[i][0];
            currentCapacity -= items[i][1];
        }

        // Otherwise take a fraction of the item
        else {
            res += (1.0 * items[i][0] / items[i][1]) * currentCapacity;

            // Knapsack is full
            break;
        }
    }

    return res;
}

int main() {
    vector<int> val = {60, 100, 120};
    vector<int> wt = {10, 20, 30};
    int capacity = 50;

    cout << fractionalKnapsack(val, wt, capacity) << endl;

    return 0;
}
```

Problem : https://cses.fi/problemset/task/2168

Given n ranges, your task is to determine for each range if it contains some other range and if some other range contains it.

Range $[a, b]$ contains range $[c, d]$ if $a \leq c$ and $d \leq b$.

	R_1	R_2	Ans
→	1	6	1
	2	4	0
	4	8	0
	3	6	0

2,4 is contained within 1,6.

Ans = 1

Check 1: Does the current range contain any other range?

→	1	6	0
	2	4	1 (contained in 1,6)
	4	8	0
	3	6	1 (contained in 4,8)

Check 2: Does the current range is contained within any other range?

For [1,6] it will be does any other range contain (1,6). No. So 0.

Output:

1 0 0 0

0 1 0 1

The idea is to sort the ranges in ascending order based on their left end. If two ranges have the same left end, prioritize the one with the larger right end. Now, we only need to check the right end for both operations:

For the "contains" operation:

- *Iterate from the right i.e from n-1 to 0. All the ranges before the current range (i.e. from i-1 to n-1) will have a left end greater than the left end of the current range. So, while traversing, if we find any right end with a value greater than the minimum right end so far, we can surely say that our current range contains that range with the minimum value of the right end as our condition is satisfied,* ***current.left ≤ prev.left and prev.min_right ≤ current.right****.*

For the "contained" operation:

- *iterate from the left i.e from 0 to n-1. All the ranges before the current range (i.e. from 0 to i-1) will have a left end lesser than the left end of the current range. So, while traversing, if we find any right end with a value lesser than the maximum right end so far, we can surely say that our current range is contained within that range with the maximum value of the right end as our condition is satisfied,* ***prev.left ≤ current.left and current.right ≤ prev.max_right****.*

AlgoZenith

```cpp
// struct to hold the range information
struct ranges {
    // The left and right ends of the range and its index in
    // the input order
    int l, r, in;

    // Overloads the < operator for sorting
    bool operator<(const ranges& other) const
    {
        // If left ends are equal, the range with larger
        // right end comes first
        if (l == other.l)
            return r > other.r;
        // Otherwise, the range with smaller left end comes
        // first
        return l < other.l;
    }
};

// Function to determine for each range if it contains some
// other range and if some other range contains it.
vector<vector<int> > checkrange(vector<vector<int> >& r,
                                int n)
{
    vector<ranges> range(n);
    vector<int> contains(n), contained(n);

    for (int i = 0; i < n; i++) {
        range[i].l = r[i][0];
        range[i].r = r[i][1];
        range[i].in = i;
    }

    // Sorts the ranges
    sort(range.begin(), range.end());

    // Checks if a range contains another
    int minEnd = 2e9;
    for (int i = n - 1; i >= 0; i--) {
        // If the right end of the current range is greater
        // than minEnd, it contains another
        if (range[i].r >= minEnd)
            contains[range[i].in] = 1;

        // Update minEnd
        minEnd = min(minEnd, range[i].r);
    }

    // Checks if a range is contained by another
    int maxEnd = 0;
    for (int i = 0; i < n; i++) {
        // If the right end of the current range is less
        // than maxEnd, it is contained by another
        if (range[i].r <= maxEnd)
            contained[range[i].in] = 1;

        // Update maxEnd
        maxEnd = max(maxEnd, range[i].r);
    }

    // Returns the contains and contained vector
    return { contains, contained };
}
```