Activity Selection in C++

```cpp
#include <iostream>
#include <algorithm>
#include <vector>
using namespace std;

class Activity {
public:
    int start;
    int finish;

    Activity(int s, int f) {
        start = s;
        finish = f;
    }
};

struct MyCmp {
    bool operator()(const Activity& a1, const
Activity& a2) const {
        return a1.finish < a2.finish;
    }
};

int maxActivity(vector<Activity>& arr) {
    sort(arr.begin(), arr.end(), MyCmp());
    int res = 1;
    int prev = 0;
    for (int curr = 1; curr < arr.size(); curr++) {
        if (arr[curr].start >= arr[prev].finish) {
            res++;
            prev = curr;
        }
    }
    return res;
}

int main() {
    vector<Activity> arr = {Activity(12, 25),
Activity(10, 20), Activity(20, 30)};
    cout << maxActivity(arr) << endl;
    return 0;
}
```

Activity Selection Problem Summary:

Given n activities with start and finish times, select the maximum number of activities that **don't overlap** and **finish earliest** (greedy approach).

📋 Input Activities (Before Sorting):

Index	Start	Finish
0	12	25
1	10	20
2	20	30

🔀 Step 1: Sort by Finish Time

Using the comparator:

return a1.finish < a2.finish;

🧾 **After Sorting:**

Index	Start	Finish
1	10	20
0	12	25
2	20	30

Sorted vector:

[{10,20}, {12,25}, {20,30}]

🧮 Step 2: Activity Selection (Greedy)

We initialize:

- res = 1 (we pick the first activity)
- prev = 0 (index of the last selected activity)

Now we iterate from curr = 1 to n-1.

➤ **Iteration Table:**

curr	Activity (start, finish)	prev	arr[curr].start >= arr[prev].finish	Action	res	prev
1	(12, 25)	0	12 >= 20 → ❌ False	Skip	1	0
2	(20, 30)	0	20 >= 20 → ✅ True	Select this	2	2

curr	Activity (start, finish)	prev	arr[curr].start >= arr[prev].finish	Action	res	prev
				activity		

✅ Final Result:

- Maximum activities: **2**
- Selected activities:
  - {10, 20}
  - {20, 30}

🖨 Output:
2

Fractional Knapsack in C++

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

class Item {
public:
    int wt, val;

     Item(int w, int v) {
        wt = w;
        val = v;
    }

    bool operator<(const Item& i) const {
        return (double)val / wt >
(double)i.val / i.wt;
    }
};

double fracKnapsack(Item arr[], int n,
int W) {
    sort(arr, arr + n);
    double res = 0.0;

    for (int i = 0; i < n; i++) {
        if (arr[i].wt <= W) {
            res += arr[i].val;
            W -= arr[i].wt;
        } else {
            res += (arr[i].val * (double)W) /
arr[i].wt;
            break;
        }
    }
    return res;
}

int main() {
    Item arr[] = {Item(10, 60), Item(40,
40), Item(20, 100), Item(30, 120)};
    int n = sizeof(arr) / sizeof(arr[0]);
    int W = 50;
    cout << fracKnapsack(arr, n, W) <<
endl;
    return 0;
}
```

Problem Summary:

You are given:

- Items with weight wt and value val
- A maximum capacity W of the knapsack
- You can **take fractions of items**

Goal: Maximize the total value in the knapsack.

📋 Input
Item arr[] = {Item(10, 60), Item(40, 40), Item(20, 100), Item(30, 120)};
int W = 50;

➤ Step 1: Calculate Value/Weight Ratio and Sort Descending

Item	Weight	Value	Value/Weight
0	10	60	6.00
1	40	40	1.00
2	20	100	5.00
3	30	120	4.00

🔀 After Sorting by Value/Weight (Descending):

Index	Weight	Value	Value/Weight
0	10	60	6.00
2	20	100	5.00
3	30	120	4.00
1	40	40	1.00

🧮 Step 2: Fill the Knapsack

Initial:
W = 50, res = 0.0

➤ Iteration Table

Iteration	Item	Weight	Value	Can Take Fully?	Action	New W	res
0	0	10	60	✅ Yes	Take full item: res += 60, W -= 10	40	60.0
1	2	20	100	✅ Yes	Take full	20	160.0

Iteration	Item	Weight	Value	Can Take Fully?	Action	New W	res
					item: res += 100, W -= 20		
2	3	30	120	❌ No	Take fraction: res += 120 * 20/30 = 80.0	0	240.0
3	1	-	-	-	Not processed (knapsack full)	0	240.0

✅ Final Output
240

240

Job Sequencing in deadline in C++

```
#include <iostream>
#include <vector>
#include <algorithm>
#include <set>

class Job {
public:
    char id;
    int deadline;
    int profit;

     Job(char id, int deadline, int
profit) {
        this->id = id;
        this->deadline = deadline;
        this->profit = profit;
    }
};

struct JobComparator {
    bool operator()(const Job& j1,
const Job& j2) {
        if (j1.profit != j2.profit)
            return j2.profit < j1.profit;
        else
            return j2.deadline <
j1.deadline;
    }
};

void
printJobScheduling(std::vector<Job>
& jobs) {
    std::sort(jobs.begin(), jobs.end(),
JobComparator());

    std::set<int> ts;
    for (int i = 0; i < jobs.size(); i++)
        ts.insert(i);

    for (const auto& job : jobs) {
        auto it =
ts.upper_bound(job.deadline - 1);
        if (it != ts.begin()) {
            --it;
            std::cout << job.id << " ";
            ts.erase(it);
        }
    }
}

int main() {
    std::vector<Job> jobs = {
        Job('a', 2, 100),
        Job('b', 1, 19),
        Job('c', 2, 27),
        Job('d', 1, 25),
        Job('e', 3, 15)
    };

    printJobScheduling(jobs);
```

Input

```
jobs = {
    Job('a', 2, 100),
    Job('b', 1, 19),
    Job('c', 2, 27),
    Job('d', 1, 25),
    Job('e', 3, 15)
}
```

➤ Step 1: Sort Jobs by Profit (Descending), Break Tie with Deadline

Job	Deadline	Profit
a	2	100
c	2	27
d	1	25
b	1	19
e	3	15

After sorting, order remains the same.

➤ Step 2: Initialize Available Time Slots

We simulate time slots using a std::set<int> ts.

ts = { 0, 1, 2, 3, 4 } // these are slot *indices*, not actual times.

We only need max_deadline = 3, so slots {0, 1, 2} are enough, but in the code ts.insert(i) for all jobs is used — let's assume the set size is sufficient.

➤ Step 3: Process Jobs One by One

We use upper_bound(job.deadline - 1) to find the latest available slot before deadline.

Job	Deadline	Profit	Find Slot ≤ Deadline - 1	Result	Scheduled?	ts After
a	2	100	upper_bound(1) → 2 → step back → 1	✅ Use slot 1	Yes	{0, 2, 3, 4}
c	2	27	upper_bound(1) → 2 → step back → 0	✅ Use slot 0	Yes	{2, 3, 4}
d	1	25	upper_bound(0) → 2 → step back → X	❌ None available	No	{2, 3, 4}
b	1	19	upper_bound(0) → 2 → step	❌ None	No	{2, 3,

```
    std::cout << std::endl;

    return 0;
}
```

Job	Deadline	Profit	Find Slot ≤ Deadline - 1	Result	Scheduled?	ts After
			back → X	available		4}
e	3	15	upper_bound(2) → 3 → step back → 2	✅ Use slot 2	Yes	{3, 4}

✅ Final Output (Jobs Scheduled)
Output: a c e

a c e